Titre: Title:	Analyse et mise en oeuvre d'égaliseurs à l'aide de processeurs configurables
Auteur: Author:	Bruno Tanguay
Date:	2005
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Tanguay, B. (2005). Analyse et mise en oeuvre d'égaliseurs à l'aide de processeurs configurables [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7537/

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

ANALYSE ET MISE EN ŒUVRE D'ÉGALISEURS À L'AIDE DE PROCESSEURS CONFIGURABLES

BRUNO TANGUAY

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE ÉLECTRIQUE)

JUILLET 2005

Your file Votre référence
ISBN: 978-0-494-16859-2
Our file Notre référence
ISBN: 978-0-494-16859-2

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ANALYSE ET MISE EN ŒUVRE D'ÉGALISEURS À L'AIDE DE PROCESSEURS CONFIGURABLES

Présenté par : TANGUAY Bruno

En vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

A été dûment accepté par le jury d'examen constitué de :

M. BOIS Guy, Ph.D., Président de jury

M. SAVARIA Yvon, Ph.D., directeur et membre

M. SAWAN Mohamad, Ph.D., co-directeur et membre

M. KHOUAS Abdelhakim, Ph.D., membre

REMERCIEMENTS

Je tiens à remercier M. Savaria, mon directeur de maîtrise pour sa disponibilité, ses conseils et son analyse qui m'ont permis de mener à terme mes travaux de recherche dans les meilleures conditions. Je tiens à souligner la contribution de M. Sawan, mon co-directeur de maîtrise, qui m'a aussi supporté dans ce projet. Je voudrais aussi souligner le support financier du CRSNG. De plus, je tiens à remercier tous les membres du groupe PROMPT-MAME (Méthodologies et Architectures de MultiÉgalisation) pour leur collaboration et les connaissances techniques dans le domaine de l'égalisation.

J'en profite pour remercier tous mes camarades du GRM (Groupe de Recherche en Microélectronique) et en particulier M. Bui pour son aide dans le domaine de la synthèse, ainsi que M. Lavigueur, M. Mbaye, M. Provost, M. Chureau, sans oublier le technicien du groupe de recherche M. Vesey et l'administrateur réseaux M. Lepage.

Je terminerai par remercier mes amis, mon frère et bien sur, mes parents qui m'ont soutenu moralement tout au long de mes études.

RÉSUMÉ

Durant les dernières années, il a été possible d'observer l'émergence de différentes normes et protocoles en télécommunication. Ces normes et protocoles ont donné lieu à une incapacité pour les systèmes sans fil de communiquer entre eux, étant donné que ceux-ci ne permettent d'utiliser qu'un nombre restreint de ces normes et protocoles. Pour pallier à ce problème, il faudrait réaliser un système radio capable d'être entièrement reconfiguré, afin de répondre aux caractéristiques spécifiques de n'importe quel type de communication. Un tel système radio flexible est souvent appelé une radio configurable par logiciel (SDR).

La réalisation d'un tel système peut s'effectuer avec trois types de plate-forme, soit des ASIC (Application Specific Integrated Circuit), des FPGA (Field Programmable Gate Array) et des DSP (Digital Signal Processor). Étant donné que l'objectif visé est celui de la flexibilité, le DSP est une solution attirante. Cependant, les performances d'une telle plate-forme sont souvent trop faibles pour leur utilisation dans ce contexte. La présente recherche vise à étudier le potentiel des technologies de processeurs configurables pour améliorer les performances dans un contexte de traitement de signal.

Dans le cadre de ce travail, l'étude a porté spécifiquement sur l'égalisation. Il s'agit d'un algorithme des plus exigeants en termes d'effort de calcul requis par les SDR. L'algorithme sélectionné est le « least-mean-square (LMS) », car cet algorithme est très utilisé pour l'égalisation étant donné sa simplicité. Deux architectures pour sa mise en œuvre ont été retenues, soit le « Linear Transversal Equalizer (LTE) » et le « Decision Feedback Equalizer (DFE) ». Le but poursuivi par cette recherche est donc de déterminer s'il est possible d'obtenir des gains en performance assez élevés pour utiliser les processeurs configurables dans un contexte SDR. Pour cette réalisation, deux technologies de processeurs configurables ont été retenues, soit celle du processeur Xtensa de Tensilica et du processeur Nios d'Altera. Ces deux technologies sont des

processeurs configurables à noyau fixe et paramétrables, qui vise une implémentation ASIC ou FPGA.

Dans un premier temps, il a fallu déterminer les techniques d'optimisation pour la conception des instructions spécialisées et les limites de l'accélération atteignable avec celles-ci. De plus, l'introduction d'instructions spécialisées dans un processeur améliore les performances, cependant, il faut être en mesure de pouvoir caractériser une instruction spécialisée en fonction du but poursuivi. Ainsi, un ensemble de métriques d'efficacité a été développé en fonction de l'objectif. Dans un autre temps, cette réalisation a permis de développer une méthodologie de conception pour les processeurs configurables avec noyau fixe et paramétrable. Finalement, plusieurs instructions spécialisées ont été conçues pour la réalisation d'égaliseurs LTE-LMS et DFE-LMS. À partir de ces implémentations, nos travaux ont permis de déterminer l'évolution de la complexité matérielle en fonction de la fréquence de l'égaliseur pour les deux technologies. Avec le processeur Xtensa, il a été possible d'obtenir un gain d'un facteur de 16 sur le temps d'exécution par rapport à une configuration de base pour la réalisation d'un égaliseur LTE-LMS et un gain d'un facteur de 21 pour un égaliseur DFE-LMS. Pour sa part, le processeur Nios a permis d'atteindre un gain d'un facteur de 47.7 en temps d'exécution pour l'égaliseur LTE-LMS et un gain d'un facteur de 64.5 pour le DFE-LMS.

ABSTRACT

Various standards in telecommunication have emerged within the last few years. The presence of these different standards has restricted communications between wireless systems, because each system can only support a limited number of standards. A radio system that is entirely reconfigurable could solve this problem by allowing a communication system to adapt its characteristics to the needs of the communications protocol. An effort to develop such flexible radio systems led to the concept of software defined radios (SDRs). Such flexible radios rely heavily on digital signal processing.

Digital signal processing cores can be implemented in three ways: as application-specific integrated circuits (ASICs), as field programmable gate arrays (FPGAs) and as digital signal processors (DSPs). ASICs offer the best performance (large bandwidth, low-power consumption, etc), but a limited flexibility. DSPs allow the most flexibility, but typically exhibit the worst performance. As for FPGAs, they offer good tradeoffs between performance and flexibility when compared to ASICs and DSPs. The aim of SDR is greater flexibility, hence DSP platforms are often preferred. However, the performance of such implementation is too low for many applications. This work aims at analyzing configurable processor technologies to improve the performance of digital signal processors.

In this work, the analysis is focused on the equalization, because it is one of the most computationally intensive tasks in a wireless system. The algorithm selected is the least-mean-square (LMS), because it is widely used for equalization due to its simplicity. Two architectures have been considered, namely the Linear Transversal Equalizer (LTE) and the Decision Feedback Equalizer (DFE). The goal of this research is therefore to determine if the gain in performance is significant enough to use configurable processors in SDR applications. Two configurable processor technologies have been selected for the purpose of this work: the Xtensa processor created by Tensilica and the Nios processor

offered by Altera. These technologies are configurable processors with a fixed and programmable core, which target both ASIC and FPGA implementations.

On one hand, it was necessary to determine the optimization techniques for specialized instructions and the limits of accelerations that can be reached with specialized instructions. The introduction of specialized instructions in a processor improves its performances, however, as this research aims at exploring their effectiveness, a set of quality metrics were developed for that task. Experience gained through this research led us to propose a methodology for developing applications on configurable processors . Finally, some specialized instructions have been created for the design of the LTE-LMS and DFE-LMS equalizers. Based on experiments reported in this thesis, we projected the hardware complexity evolution for the two considered technologies. By using the Xtensa processor, it was possible to achieve an acceleration factor of 17 for an LTE-LMS equalizer, when compared to a basic configuration, and an acceleration factor of 22 for a DFE-LMS equalizer. On the other hand, the Nios processor allowed to obtain an acceleration factor of 47.7 for an LTE-LMS equalizer, and an acceleration factor of 64.5 for a DFE-LMS equalizer.

TABLE DES MATIÈRES

LISTES DES FIGURES

LISTE DES TABLEAUX

SIGLES ET ABRÉVIATIONS

ALU	**A**rithmetic and **L**ogic **U**nit
AM	**A**mplitude **M**odulation
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
ASIP	**A**pplication **S**pecific **I**nstruction-Set **P**rocessor
BPSK	**B**inary **P**hase **S**hift **K**eying
CDMA	**C**ode **D**ivision **M**ultiple **A**ccess
CIC	**C**ascaded **I**ntegrator **C**omb
CPU	**C**entral **P**rocessing **U**nit
DDS	**D**irect **D**igital **S**ynthesis
DFE	**D**ecision **F**eedback **E**qualizer
DFG	**D**ata**f**low **G**raph
DSP	**D**igital **S**ignal **P**rocessor
DSSS	**D**irect-**S**equence **S**pread-**S**pectrum
EDGE	**E**nhanced **D**ata Rates for **G**SM **E**volution
EPLRS	**E**nhanced **P**osition **L**ocation **R**eporting **S**ystem
FBE	**F**eed**B**ack **E**qualizer
FDMA	**F**requency **D**ivision **M**ultiple **A**ccess
FFE	**F**eed**F**oward **E**qualizer
FFT	**F**ast **F**ourier **T**ransform
FH	**F**requency **H**opping
FIR	**F**inite **I**mpulse **R**esponse
FM	**F**requency **M**odulation
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSE	**F**ractionally **S**paced **E**qualizer

GERMS	**G**o **E**rase **R**elocate **M**emory **S**end
GNU	**G**NU's **N**ot **U**nix (acronyme récursif)
GPS	**G**lobal **P**ositioning **S**ystem
GRM	**G**roupe de **R**echerche en **M**icroélectronique
GSM	**G**lobal **S**ystem for **M**obile Communication
HDL	**H**ardware **D**escription **L**anguage
HF	**H**igh **F**requency
IFFT	**I**nverse **F**ast **F**ourier **T**ransform
IMT-2000	**I**nternational **M**obile **T**elecommunication by 2000
IIR	**I**nfinite **I**mpulse **R**esponse
IS-XX	**I**nterim **S**tandard **XX** (XX = 54, 95, 136)
IS	**I**nstructions **S**pécialisées
ISS	**I**nstruction-**S**et **S**imulator
ISI	**I**nter-**S**ymbol **I**nterference
ITU	**I**nternational **T**elecommunications **U**nion
JTIDS	**J**oint **T**actical **I**nformation **D**istribution **S**ystem
LC	**L**ogic **C**ells
LISA	**L**anguage for **I**nstruction **S**et **A**rchitecture
LMS	**L**east **M**eans **S**quare
LSB	**L**east **S**ignificant **B**it
LTE	**L**inear **T**ransversal **E**qualizer
LUT	**L**ook-**U**p **T**able
MAC	**M**ultiplieur **A**ccumulateur (**M**ultiply-**A**ccumulate)
MBPS	**M**ega**B**its **P**er **S**econd
MC	**M**ulti **C**arriers
MIMD	**M**ultiple **I**nstruction **M**ultiple **D**ata
MIMO	**M**ultiple-**I**nput **M**ultiple-**O**utput
MISO	**M**ultiple-**I**nput **S**ingle-**O**utput
MSB	**M**ost **S**ignificant **B**it

NCO	**N**umerically **C**ontrolled **O**scillator
PDC	**P**ersonal **D**igital **C**ellular
PSK	**P**hase **S**hift **K**eying
QAM	**Q**uadrature **A**mplitude **M**odulation
QPSK	**Q**uadrature **P**hase **S**hift **K**eying
RAM	**R**andom **A**ccess **M**emory
RISC	**R**educed **I**nstruction-**S**et **C**omputer
RLS	**R**ecursive **L**east-**S**quares
ROM	**R**ead-**O**nly **M**emory
RTL	**R**egister **T**ransfert **L**evel
SATCOM	**Sat**ellite **Com**munication
SDR	**S**oftware **D**efined **R**adio
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SINCGARS	**S**ingle **C**hannel **G**round and **A**irborne **R**adio **S**ystem
SOPC	**S**ystem-**O**n-**P**rogrammable-**C**hip
TDMA	**T**ime **D**ivision **M**ultiple **A**ccess
TIE	**T**ensilica **I**nstruction **E**xtension
UMTS	**U**niversal **M**obile **T**elecommunication **S**ystem
VHDL	**VHSIC HDL**
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
VLIW	**V**ery **L**ong **I**nstruction **W**ord
XLMI	**X**tensa **L**ocal **M**emory **I**nterface

CHAPITRE 1

INTRODUCTION

La demande grandissante pour les communications sans fil a donné naissance à différentes normes et protocoles de télécommunication à travers le monde. Ces normes et protocoles possèdent des caractéristiques propres en termes de type d'accès, de modulation, de fréquence porteuse, d'encodage et de bande passante. Ainsi, les dispositifs actuels de communication sans fil sont optimisés principalement pour une seule ou quelques-uns de ces normes. À titre d'exemple de la diversité des normes et protocoles existants, le tableau 1.1 présente leurs caractéristiques en terme de porteuse, de bande passante et de type de modulation [20][54].

Tableau 1.1 : Diversité des normes et protocoles de communication

Standard	Bande fréquentielle	Bande passante du canal	Modulation
FH HF	2 à 20 MHz	3 kHz	AM/FM
SINCGARS	30 à 88 MHz	25 kHz	FH (100 h/s)
Have Quick	225 à 400 MHz	25 kHz	FH
EPLRS	420 à 450 MHz	5 MHz	DSSS/FH
JTIDS	960 à 1310 MHz	6 MHz	DSSS/FH
GPS	1.5 GHz	10 MHz	DSSS
SATCOM	7 GHz	Variable	Variable
IS-95	800 MHz, 1.9 GHz	1.25 MHz	DSSS/CDMA
IS-54/136	800 MHz	30 kHz	TDMA
ITU 3G/IMT-2000	1.8 à 2.1 GHz	5,10, 15 MHz	DSSS/CDMA
GSM 900,1800	900 MHz, 1800 MHz	25 kHz	FDMA/TDMA
PDC	800 MHz/ 1500 MHz	25 kHz	TDMA (π/4-DQPSK)

Un problème important qui découle de l'émergence de toutes ces différentes normes est la compatibilité des dispositifs de télécommunication entre eux. En fait, chaque dispositif est conçu pour permettre l'utilisation d'un nombre restreint de normes. Il en résulte une incapacité entre les systèmes sans fil de communiquer entre eux. La solution à un tel problème serait un système radio capable d'être entièrement reconfiguré pour permettre à celui-ci de répondre aux caractéristiques spécifiques de n'importe quel type de communication. Avec une telle solution, la même plateforme matérielle pourrait être reconfigurée pour permettre de supporter différentes fonctionnalités à différents moments.

Ainsi, la tendance actuelle dans le domaine des télécommunications est d'effectuer une migration des transcepteurs (transmetteur-récepteur) dédiés vers une radio flexible capable de supporter de multiples modulations et de multiples normes et protocoles. Le but ultime est la réalisation d'une radio capable de supporter tous les types possibles de communication sans fil (cellulaire, militaire, urgence, satellite et autre), ce qui implique le transport de n'importe quel type de donnée : audio, vidéo, fax et données [36]. En d'autres mots, pour surmonter le problème de compatibilité, plusieurs groupes de recherche à travers le monde sont présentement en train d'investiguer le développement d'une radio universelle capable d'opérer avec n'importe quel système de communication. Une telle réalisation permettrait d'implémenter de nouvelles normes et protocoles par une simple re-programmation. Ceci permettrait aussi de réduire le coût de production et de vérification des dispositifs de communication, et d'implanter de façon rapide et facile une mise à jour du système. Dans la littérature, ce concept de radio re-configurable est souvent appelé « *Software Defined Radio* » (SDR), mais le terme « *Software Radio* » est utilisé fréquemment.

Dans un contexte SDR, le but ultime est de réaliser la majeure partie de la fonctionnalité dans le domaine numérique. Ceci se traduit à rapprocher le plus possible le convertisseur analogique à numérique de l'antenne [8][36][37]. L'essence de la SDR est de réaliser une plate-forme matérielle qui puisse être modifiée à l'aide de logiciel afin de redéfinir ses

paramètres, comme sa bande fréquentielle, son débit d'information et son type de modulation. Ainsi, la chaîne de traitement numérique d'une SDR doit être aussi flexible que possible [16][47]. Dans la suite de ce chapitre, la problématique du sujet de recherche sera explicitée, suivie des objectifs de cette recherche et de la méthodologie. Enfin, le plan du mémoire sera détaillé.

1.1 Problématique

La chaîne de traitement numérique peut être réalisée avec des ASIC, des FPGA, des DSP ou avec une combinaison de ces trois types de dispositifs. Les ASIC offrent la meilleure performance, mais peu de flexibilité. Pour leur part, les DSP offrent la plus grande flexibilité, mais offrent généralement les pires performances. En ce qui concerne les FPGA, ceux-ci se retrouvent entre les ASIC et les DSP pour leur flexibilité et leur performance. De plus, il est à spécifier qu'au cours du traitement numérique (à partir de l'antenne) d'une SDR, le débit d'information diminue, par contre, la complexité de l'algorithme de traitement augmente. Idéalement, une chaîne de traitement numérique d'une SDR serait constituée d'un ou de plusieurs DSP. Ainsi, une reconfiguration du système ne nécessiterait qu'un rechargement du code qui s'exécute sur le DSP.

Par conséquent, dans la partie avant de la chaîne numérique, il faut utiliser des ASIC ou des FPGA étant donné que le traitement s'effectue à haut débit. Par contre, les DSP peuvent être utilisés en fin de traitement où le débit d'information est moins élevé, et ce, afin d'accroître la flexibilité. Comme mentionné précédemment, ce qui limite l'utilisation des DSP c'est leur rapidité de traitement. Il serait donc intéressant d'étudier s'il ne serait pas possible d'augmenter les performances des DSP. Pour ce faire, la présente étude est en fait axée vers la conception d'instructions spécialisées à partir de processeurs configurables.

1.2 Objectif

L'objectif de ce travail est d'étudier les processeurs configurables pour leur utilisation dans un contexte SDR. Cependant, l'étude s'est concentrée sur l'égalisation en ce qui a trait la SDR, car c'est l'un des algorithmes les plus exigeants en termes d'effort de calcul. Dans le contexte de ce travail, l'algorithme sélectionné est le « *Least-Mean-Square* (LMS) ». Cet algorithme est très utilisé étant donné sa simplicité. Deux architectures ont été retenues, soit le « *Linear Transversal Equalizer* (LTE) » et le « *Decision Feedback Equalizer* (DFE) ». Le but poursuivi par cette recherche est donc de déterminer s'il est possible d'obtenir des gains en performance assez élevés pour utiliser les processeurs configurables dans un contexte SDR, de déterminer la provenance et les limitations de ces gains et comment caractériser une instruction spécialisée. Il est à noter que dans le cadre de ce mémoire, le terme performance est généralement utilisé pour désigner le nombre d'opérations effectuées par unité de temps. Par conséquent, une augmentation des performances implique une réduction du temps d'exécution pour l'accomplissement d'un nombre constant d'opérations.

1.3 Méthodologie

Afin d'atteindre les objectifs fixés, il est d'abord nécessaire de se familiariser avec les processeurs configurables. Pour le cadre de la recherche, deux processeurs configurables ont été retenus, soit le processeur Xtensa de la société Tensilica et le processeur Nios de la société Altera. Le premier processeur vise normalement une implémentation ASIC, alors que le second vise une implémentation FPGA. La technologie du processeur Xtensa est relativement mature et offre une vaste gamme d'outils qui permettent de concevoir des instructions spécialisées. Pour sa part, le processeur Nios est fortement utilisé en prototypage rapide et il est facile d'accès. Ce processeur offre l'opportunité d'intégrer des

instructions spécialisées au noyau, cependant, l'ensemble des outils n'est pas axé sur cet aspect du processeur.

En second lieu, des égaliseurs LTE et DFE seront réalisés avec une plate-forme ASIC et à l'aide du processeur configurable Xtensa. Troisièmement, des égaliseurs LTE et DFE seront développés avec une plate-forme FPGA et à l'aide du processeur configurable Nios. À partir de cette expérience, une méthodologie de conception pour exploiter les processeurs configurables sera développée. Nous proposons aussi un ensemble de métriques de qualité afin de mesurer l'efficacité des instructions spécialisées et nous étudions les limites de l'accélération atteignable avec des instructions spécialisées.

1.4 Organisation du mémoire

Le prochain chapitre du mémoire traite des processeurs configurables. Il se concentre principalement sur les technologies utilisées. Ce chapitre décrit aussi quelques autres processeurs disponibles sur le marché. Le troisième chapitre discute des techniques d'optimisation pour la conception de processeurs configurables. Il présente dans un deuxième temps les métriques de qualités des instructions spécialisées en fonction de l'objectif poursuivi. Ce chapitre se termine par une exploration des limites de l'accélération atteignable avec des instructions spécialisées. Le quatrième chapitre de ce mémoire effectue un survol de la méthodologie de conception avec les processeurs configurables, que celle-ci soit automatique ou manuelle. Le cinquième chapitre traite des résultats obtenus avec les deux processeurs configurables pour l'implémentation des deux égaliseurs. Les détails de l'implémentation des instructions spécialisées sont d'abord présentés. Ensuite, les résultats obtenus avec le processeur Xtensa sont exposés, suivi par les résultats obtenus avec le processeur Nios.

CHAPITRE 2

PROCESSEURS CONFIGURABLES

La technologie des processeurs configurables a gagné considérablement en intérêt depuis les dernières années. Le présent chapitre est dédié à une synthèse de la technologie actuelle. La première section de ce chapitre est consacrée aux motivations relatives à une technologie de processeur configurable. Dans un deuxième temps, un survol des différentes technologies sera effectué. Ce survol s'amorcera avec la technologie du processeur Xtensa. Il se poursuivra avec la technologie du processeur Nios, pour se terminer par d'autres technologies comme Lisatek et Target Compiler.

2.1 Motivations

Comme mentionné précédemment, les ASIC sont généralement les plateformes les plus performantes en termes de vitesse d'exécution et les DSP sont les plus flexibles au niveau de la fonctionnalité. Pour obtenir un compromis entre ces deux extrêmes, il est possible d'utiliser des processeurs configurables. L'avantage d'une telle technologie est de conserver une flexibilité au niveau de l'application tout en permettant d'atteindre les objectifs de performance. Cette flexibilité permet d'effectuer des changements de fonctionnalité sans avoir à payer le prix des modifications d'un ASIC. De ce fait, une simple modification logicielle permet des modifications au niveau de la fonctionnalité.

Ces dispositifs sont des processeurs généraux desquels il est possible de modifier la configuration en ce qui concerne le type d'antémémoire, les unités fonctionnelles présentes et le jeu d'instruction supporté. Ainsi, ceux-ci permettent d'obtenir de meilleures performances en ajoutant du matériel au processeur afin d'optimiser le temps d'exécution du programme. Ce type de processeurs permet d'adapter son jeu d'instructions à une application donnée en tirant profit des caractéristiques propres à

l'application. Ainsi, cette technologie est en fait un moyen de rencontrer les objectifs de performance, de coût matériel et de consommation de puissance. Dans la littérature, le terme ASIP (« *Application Specific Instruction-set Processor* ») est souvent employé.

Présentement, plusieurs processeurs configurables sont disponibles sur le marché, citons comme exemple le processeur Xtensa de la compagnie Tensilica, le processeur NIOS d'Altera, le processeur ARC tangent, le processeur LisaTek et le processeur Target Compiler. Chacun d'eux offre différentes options en vue de la conception de processeurs spécialisés. Il existe pour ces différents processeurs un ensemble d'outils pour automatiser la conception. Ainsi, le processus de conception peut être accéléré par ces outils. Ils permettent, tout dépendant de la technologie spécifique, de générer le processeur, de générer des instructions spécialisées (IS), de simuler l'exécution d'un programme sur un processeur spécialisé (profilage) et de développer du logiciel.

Les ASIC sont optimisés au niveau des performances, de l'aire et de la consommation de puissance, cependant, le processus de mise en marché peut être très long. Ainsi, le temps de mise en marché peut être problématique. Les outils de conception automatisée des ASIP permettent de réduire ce temps de mise en marché. De plus, le processus de développement pour du logiciel est plus rapide que celui requis pour du matériel. S'il faut effectuer un changement majeur dans la fonctionnalité, cela nécessite des coûts exorbitants dans le cas de la technologie ASIC, alors qu'une simple reprogrammation peut suffire avec une technologie ASIP.

Le principal objectif de conception des ASIP est l'augmentation de l'efficacité du système (coût matériel, complexité du code, performance, consommation de puissance). En fait lorsque des instructions spécialisées sont conçues, une nouvelle instruction vient remplacer un ensemble d'instructions, ce qui permet normalement d'augmenter les performances et de réduire la complexité du code. De plus, il est possible de réduire la consommation de puissance du processeur. En effet, si un ensemble d'instructions spécialisées accélèrent l'exécution d'un programme par un facteur 4 mais n'en

n'augmente la puissance consommée que par deux, la consommation de puissance se trouve alors diminuée de moitié pour une même performance.

2.2 Processeur Xtensa

Une partie de l'étude effectuée dans ce mémoire a été réalisée sur le processeur configurable Xtensa de la compagnie Tensilica. Cette section est consacrée à faire un bref survol de cette technologie. Pour en connaître davantage sur ce processeur, de plus amples détails sont dévoilés dans [52].

2.2.1 Résumé de la technologie

Ce processeur est en fait un microprocesseur RISC (« *Reduced Instruction-Set Computer* ») standard auquel il est possible d'ajouter des options prédéfinies ou même des instructions. La façon de procéder est de sélectionner les options désirées au sein du processeur générateur, qui se trouve à être un outil qui génère d'autres outils logiciels et matériels pour le processeur généré. Le générateur de processeur fournit le code RTL pour la synthèse matérielle, des outils de simulation ainsi que des outils logiciels qui permettent de compiler son code et de le profiler pour ce processeur spécifique. Une fois le processeur généré, il est possible d'exécuter le programme afin de déterminer combien de cycles nécessitent différentes sections du programme. À partir de ces données, il est possible d'ajouter des instructions pour accélérer les sections qui consomment le plus de cycles d'exécution.

2.2.2 Architecture du processeur

Ce processeur est un RISC 32 bits pipeliné sur 5 étages. Les 5 étages du pipeline sont illustrés à la figure 2.1. Le premier étage est celui qui va chercher les instructions en

mémoire (I). Le second étage décode l'instruction et recherche les registres visés par cette instruction (R). Le troisième étage exécute l'instruction et effectue la résolution des adresses pour les instructions de chargement de données (E). Le quatrième étage est celui qui effectue les accès en mémoire ou qui complète les branchements (M). Le dernier étage retourne les nouvelles valeurs dans les registres concernés (W). Si une instruction spécialisée est ajoutée au processeur, le décodeur sera mis à jour automatiquement pour reconnaître cette nouvelle instruction. De plus, les registres spécialisés seront ajoutés au second étage du pipeline et seront sélectionnés lorsque l'instruction spécialisée est appelée. L'exécution de l'instruction spécialisée est effectuée au troisième étage du pipeline. Comme illustré à la figure 2.1, l'exécution d'une instruction spécialisée peut être répartie sur deux étages du pipeline (E & M) pour les instructions spécialisées qui possèdent un chemin critique plus long qu'une période d'horloge. En ce qui concerne l'arrêt du pipeline, il faut mentionner que toute instruction suivant un chargement de données qui utilise le résultat du chargement (aléa de lecture après écriture) nécessite un arrêt d'un cycle. De plus, les instructions de branchement sont résolues à l'étage de l'exécution, il faut par conséquent deux cycles afin de résoudre l'adresse de la nouvelle instruction.

Le jeu d'instructions de base du processeur Xtensa est composé d'environ 80 instructions. Celui-ci est composé des instructions de base qui effectuent les chargements et les enregistrements des données, les instructions de branchement, les instructions arithmétiques et logiques, les instructions de déplacement de données et les instructions de contrôle du processeur. Les instructions de contrôle du processeur sont utilisées pour écrire et lire des registres spécialisés et pour se synchroniser. Toutes les instructions sont encodées sur 16 ou 24 bits. Un encodage de 16 bits est utilisé pour les instructions les plus utilisées. Cette façon de faire permet de réduire la taille du code. À chaque recherche d'instruction, une valeur de 32 bits est recherchée dans la mémoire et 16 ou 24 bits d'instruction y sont extraits. Ainsi, des instructions de 16 et 24 bits peuvent être entremêlés sans problème.

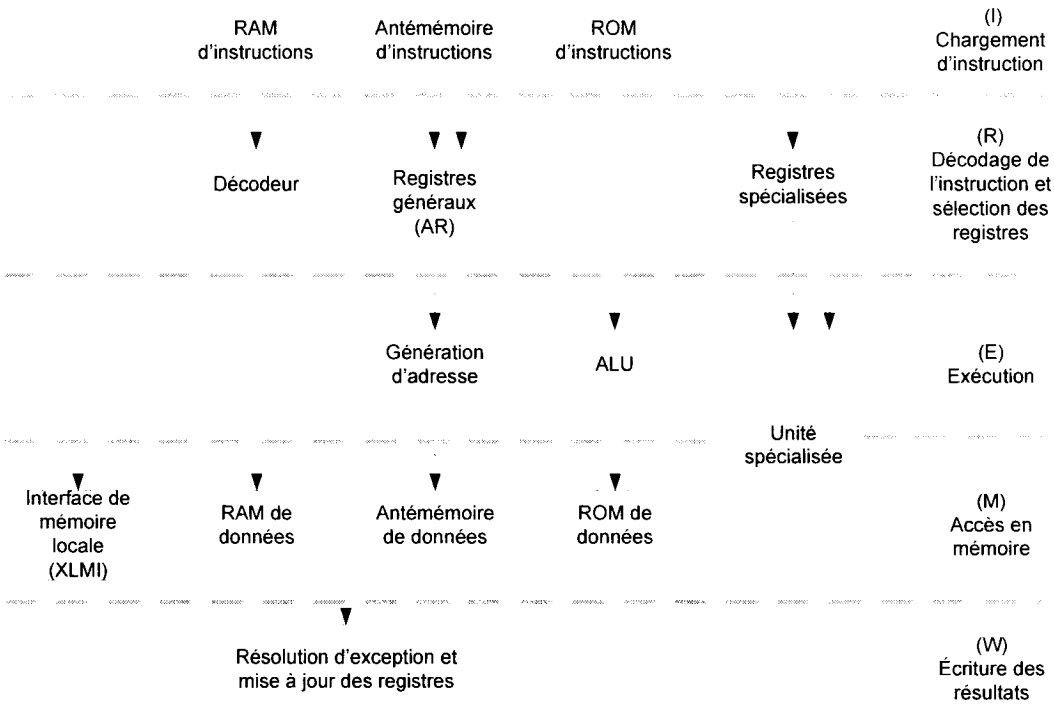

Figure 2.1 : Pipeline du processeur Xtensa

Ce processeur possède 16 registres logiques et 32 ou 64 registres physiques généraux. Les registres utilisent un mécanisme de fenêtre coulissante, ce mécanisme n'est pas perceptible par le programmeur. Il est possible de faire des incréments de 4, 8 ou 12 registres sur les registres physiques, ceci est déterminé par la fonction appelante. Cette façon de faire permet d'accélérer les appels de fonction, car ce mécanisme évite la sauvegarde complète des registres.

2.2.3 Options de configuration

À la base, plusieurs options peuvent être ajoutées. Parmi ces principales options, il est possible d'ajouter des unités opératives, comme les multiplieurs (16 ou 32 bits), un MAC (Multiplieur–Accumulateur) de 16 bits, une unité arithmétique virgule flottante et une unité Vectra DSP SIMD (« *Single Instruction Multiple Data* »). De plus, ce processeur

peut contenir 32 ou 64 registres physiques, l'ordonnancement de mémoire peut être gros-boutiste (« *big-endian* ») ou petit-boutiste (« *little-endian* »). L'une des configurations intéressantes de ce processeur est de pouvoir activer un mécanisme de bouclage automatique sans payer de cycle additionnel (« *Zero-Overhead Loop Instructions* »). En ce qui concerne les interruptions, le processeur peut gérer jusqu'à 32 interruptions externes, 6 niveaux de priorités d'interruption, des interruptions non-masquable et jusqu'à 3 compteurs de temps (« *timers* ») à 32 bits. L'encodage des instructions sur 16 bits pour augmenter la densité du code peut être désactivé, c'est-à-dire que les instructions seront toutes encodées sur 24 bits si cette option est désactivée. Les interfaces du processeur sont aussi configurables. Par conséquent, les accès à la mémoire principale du système peuvent être sur 32, 64 ou 128 bits.

En ce qui concerne l'antémémoire, plusieurs options permettent d'adapter celle-ci à l'application visée. Parmi ces options, la taille des lignes de la cache peut être de 16, 32 ou 64 octets. Le degré d'associativité pour l'antémémoire peut être par 1, 2, 3 ou 4 blocs par ensemble (« 1, 2, 3, 4-*way set associative* »). La politique d'écriture pour l'antémémoire de données peut être l'écriture simultanée (« *Write-through* ») ou la réécriture (« *Write-Back* »). La taille de l'antémémoire pour les données et les instructions est variable. Le tableau 2.1 montre les possibilités. À cette mémoire, il est aussi envisageable d'ajouter de la mémoire RAM ou/et ROM de taille variable. Cette taille peut prendre les valeurs suivantes 0, 512, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K et 256K octets.

Tableau 2.1 : Taille de l'antémémoire pour le processeur Xtensa

Taille de l'antémémoire	1 bloc par ensemble	2 blocs par ensemble	3 blocs par ensemble	4 blocs par ensemble
0/1Ko/2Ko/4Ko/8Ko/16Ko/32Ko	X	X		X
0/1.5Ko/3Ko/6Ko/12Ko/24Ko			X	

2.2.4 Instructions spécialisées

La réalisation d'instruction spécialisée est effectuée à l'aide d'un langage propriétaire qui permet des extensions au jeu d'instruction appelées TIE (« *Tensilica Instruction Extension* »). Le langage qui permet d'exprimer des TIE est similaire à du Verilog. Ce langage permet donc de définir la fonctionnalité des instructions spécialisées, l'« opcode » de l'instruction, les opérandes et la sémantique des instructions. De plus, le nombre de registres spécialisés et la taille de ceux-ci sont définis à l'aide de TIE. Étant donné que l'interface est configurable, des instructions de chargement et d'enregistrement de données de 64/128 bits peuvent aussi être ajoutées au processeur. Ceci permet d'augmenter grandement la bande passante entre la mémoire et le processeur lorsque l'application le nécessite. Toutes les nouvelles instructions peuvent être exécutées sur 2 cycles comme mentionné précédemment (voir section 2.2.2). Lorsqu'une instruction est planifiée sur 2 cycles d'horloge, le matériel généré pour cette instruction est automatiquement pipeliné. Ainsi, ce langage permet d'intégrer de nouvelles instructions au processeur de base, par conséquent, il est alors possible de paralléliser certains traitements afin d'accélérer l'exécution du programme.

Pour générer des instructions, il suffit d'écrire le code TIE correspondant à la nouvelle fonctionnalité. Le code TIE est ensuite compilé à l'aide d'un outil qui génère du code Verilog correspondant et du code logiciel qui permet de profiler avec ces nouvelles instructions. Cette façon de procéder permet d'effectuer plusieurs itérations sans avoir à re-générer le processeur de base. Lorsque le générateur de processeur Xtensa intègre les nouvelles instructions au corps, il utilise l'information de contrôle générée par le compilateur TIE pour coupler la nouvelle unité fonctionnelle au pipeline du processeur. Ainsi, les mécanismes de déviation (« *bypass* ») et de détection de blocage (« *interlock detection* ») sont automatiquement intégrés.

2.2.5 Outils de conception

La technologie Xtensa est en réalité un ensemble d'outils qui permet la conception rapide de processeurs spécialisés. Ces outils sont divisés principalement en trois catégories, soit ceux pour la conception matérielle, ceux pour le développement logiciel et ceux pour la modélisation du système. Parmi les outils de conception matérielle, le générateur de processeur Xtensa fournit le code RTL synthétisable du processeur ainsi que des scripts pour des outils utilisés fréquemment pour la synthèse et le placement/routage. Pour le développement logiciel, un ensemble d'outils logiciel adapté de GNU, soit un compilateur, un assembleur, un profileur, un éditeur de lien (« *linker* ») et un débogueur est fourni avec chaque configuration. Avec ces mêmes outils, des bibliothèques en C, un compilateur Xtensa (XCC) et un ensemble de fonctions de support pour différents OS sont à la disposition du concepteur logiciel. Parmi les outils pour la modélisation du système, Tensilica fournit un simulateur de jeu d'instruction (« *Instruction Set Simulator* », ISS), un environnement de simulation système (XTMP) et un modèle fonctionnel de bus.

Cette technologie cible avant tout une implémentation ASIC, par contre, il est aussi possible de faire fonctionner ce processeur configurable sur un FPGA.

2.3 Processeur Nios

La technologie précédemment discutée permet de réaliser la conception d'un ASIP dédié à une plate-forme ASIC. Cependant, il existe sur le marché des processeurs configurables dont le noyau est sous forme logicielle (« *soft core* ») et qui cible des plates-formes FPGA. La technologie du processeur Nios est en fait de ce type [2].

2.3.1 Résumé de la technologie

Ce processeur est en fait un microprocesseur RISC auquel il est possible d'ajouter des options et des périphériques. Dans le CPU, il est possible d'intégrer une unité spécialisée

en parallèle à l'ALU conventionnel. La façon de procéder est de sélectionner les options désirées au sein de l'outil « *SOPC builder* ». Cet outil fournit le code VHDL pour la synthèse matérielle, des bibliothèques pour le développement logiciel et un modèle de simulation pour Modelsim. Une fois le processeur généré, il est possible de compiler le processeur pour le FPGA cible et de télécharger la configuration sur la plate-forme. Ce processeur peut intégrer un programme (« *GERMS monitor* ») qui permet de télécharger un nouveau programme lors de la mise en route du processeur.

2.3.2 Architecture du processeur

C'est un processeur RISC pipeliné sur 5 étages qui exécute une seule instruction par cycle (« *single-issue* »). Ce processeur a la particularité de pouvoir fonctionner sur des données de 16 ou 32 bits tout dépendant de la configuration. Son jeu d'instruction est encodé sur 16 bits pour les deux variantes du processeur. L'architecture de ce processeur est du type Harvard, donc un bus pour les données et un autre pour les instructions. Ces deux bus sont de type Avalon. Il est possible d'avoir jusqu'à 512 registres internes à usage général dans ce processeur. La taille du fenêtrage de ces registres est toujours de 32 registres. Ce fenêtrage est coulissant avec une granularité de déplacement de 16 registres. Ce mécanisme est utilisé pour accélérer les appels de fonction. L'exécution de la plupart des instructions est effectuée sur un seul cycle. Cependant, le Nios utilise un décaleur logique barrillet (« *barrel-shifter* ») qui exécute toutes les instructions de décalages en deux cycles, et ce, peu importe la distance du décalage. Il est à spécifier que ce processeur est de type petit-boutiste (« *little-endian* »).

2.3.3 Options de configuration

L'une des premières options de configuration du Nios est de déterminer la largeur des données, soit de 16 ou 32 bits. Toutes les instructions sont de 16 bits, ce qui réduit la largeur du code et la bande passante de la mémoire instruction. Il est possible d'intégrer

128, 256 ou 512 registres physiques. De plus, il est possible d'ajouter de l'antémémoire et la taille de celle-ci est configurable pour les données, comme pour les instructions (seulement disponible pour le Nios 32-bit). La taille de la mémoire cache peut être de 1K, 2K, 4K, 8K ou 16Koctets. L'antémémoire est du type d'association directe (« *direct-map* »). Celle-ci peut être désactivée de façon logicielle.

Tableau 2.2 : Détails du CPU selon la variante du processeur Nios

Détails du CPU	Nios 32-bit	Nios 16-bit
Taille du bus de données (bits)	**32**	**16**
Taille de l'ALU (bits)	**32**	**16**
Taille des registres internes (bits)	**32**	**16**
Taille du bus d'adresse (bits)	**32**	**16**
Taille des instructions (bits)	**16**	**16**

En ce qui concerne l'opération de multiplication, il est possible de choisir parmi trois options. La première option consiste à intégrer une unité de multiplication matérielle, qui prend de 1 à 3 coups d'horloge tout dépendant du FPGA. La seconde option est l'intégration d'une unité MSTEP qui fournit un résultat partiel à chaque deux cycles. Pour cette dernière, il faut environ 16 appels consécutifs pour réaliser une multiplication 16x16. Cette unité nécessite moins de 5% de la logique du CPU. Cette option n'est disponible que pour le Nios 32-bit. La dernière option est l'utilisation d'une multiplication logicielle, i.e. avec une séquence d'addition et de décalage, ce qui prend considérablement de temps à s'exécuter.

2.3.4 Instructions spécialisées

Le concepteur d'un processeur Nios peut accélérer des sections de code critique en ajoutant des instructions spécialisées au noyau. Les instructions spécialisées peuvent être exécutées sur un seul cycle (combinatoire) ou sur plusieurs (séquentiel). De plus, la

logique de l'unité spécialisée peut accéder à la mémoire et/ou de la logique externe au Nios. Selon la spécification du Nios [2], il est possible d'intégrer jusqu'à 5 instructions spécialisées au noyau. Il n'y a aucune restriction sur la fonctionnalité de l'unité spécialisée tant que l'interface est appropriée. L'interface est constituée au minimum de deux entrées et d'un résultat de la taille du CPU (16 ou 32 bits). Le reste de l'interface est utilisé pour des signaux de contrôle (clk, clk_en, reset, start) et pour d'autres entrées laissées au concepteur (ports externes, prefix), ceux-ci sont facultatifs.

Chaque instruction spécialisée peut être définie en VHDL, en Verilog HDL ou en avec d'autres méthodes (interconnexions de modules). Une fois générée, une instruction spécialisée est intégrée au noyau à l'aide de « *SOPC builder* ». Cet outil génère automatiquement une trousse de développement logicielle inhérente au processeur. Cette trousse contient des macros pour accéder aux instructions spécialisées.

2.3.5 Outils de conception

Le principal outil de conception est *Quartus II*, qui se trouve à être un environnement de synthèse pour les FPGA d'Altera. Cet environnement permet d'interconnecter et de créer les modules qui seront aux alentours du processeur Nios, ainsi que d'effectuer l'assignation des broches du FPGA. Le principal outil qui sert à la génération du processeur est le « *SOPC builder* ». Celui-ci permet de configurer le processeur selon les besoins de l'application et d'ajouter une unité spécialisée en parallèle à l'ALU. C'est ce même outil qui génère la trousse de développement logicielle, le code VHDL (ou Verilog) du processeur et l'environnement de simulation pour *Modelsim*.

La trousse de développement logiciel contient des bibliothèques qui permettent d'utiliser les périphériques et des fonctions courantes. Tout le code logiciel est compilé à l'aide d'un environnement GNU (Compilateur C/C++, débogueur, éditeur de lien, assembleur). De plus, il est possible d'utiliser un profileur fourni à cet effet (gprofiler). Cependant, celui-ci est loin d'être précis au cycle près.

2.4 Autres processeurs configurables

2.4.1 LisaTek

La technologie LisaTek a été développée à l'université d'Aachen en Allemagne. Maintenant, elle est un produit commercial de la compagnie Coware [33]. Cette technologie est en fait un outil pour automatiser la conception d'un processeur embarqué. La philosophie de base est axée sur la description de l'architecture du processeur à partir du langage LISA (« *Language for Instruction Set Architecture* »). Ce langage est en fait un langage de description matérielle comme le VHDL et le Verilog, à l'exception qu'il est spécialisé dans l'exploration architecturale de processeurs. Celui-ci sert à décrire les architectures programmables, les périphériques et les interfaces du processeur. Ce langage offre une grande flexibilité pour la description du jeu d'instruction, il peut intégrer des instructions de type SIMD, MIMD et VLIW. Celui-ci permet aussi de modéliser des pipelines et des mécanismes de contrôle complexes. Cette technologie offre une flexibilité au niveau du noyau du processeur, ce que les processeurs Xtensa et Nios n'offrent pas.

Le LISA supporte plusieurs modèles [24][41][46]:

- Modèle des mémoires : Ce modèle énumère tous les registres et mémoires du système avec leurs caractéristiques pour la taille des mots, leurs adresses et les références.
- Modèle des ressources : Ce modèle décrit les ressources matérielles utilisées par chacune des instructions dans le processeur, comme les registres, les unités fonctionnelles. Celui-ci reproduit les propriétés de la structure matérielle.
- Modèle du jeu d'instruction : Ce modèle identifie toutes les combinaisons possibles d'opérations matérielles et les opérandes admissibles. Celui-ci comprend la syntaxe assembleur, l'encodage de l'instruction, la spécification sur les opérandes valides et les modes d'adressage.

- Modèle comportemental : Ce modèle permet d'abstraire les activités des structures matérielles pour la simulation. C'est en fait un modèle de simulation pour les unités fonctionnelles.
- Modèle temporel : Ce modèle spécifie la séquence d'activation des opérations matérielles et des unités.
- Modèle microarchitecture : Ce modèle permet de grouper des opérations matérielles en unité fonctionnelle. Celle-ci contient l'implémentation exacte du composant sous forme structurelle.

La méthodologie de conception avec ce type de technologie commence a priori par une description de l'architecture du processeur avec le langage LISA. Cette méthodologie est séparée en deux phases principalement, soit une phase d'exploration et une phase d'implémentation. La phase d'exploration consiste à générer automatiquement des outils de développement logiciel qui correspondent à la description du processeur. Les outils de développement logiciel comprennent un compilateur, un assembleur, un éditeur de liens et un simulateur. Ainsi, il est possible d'essayer différentes configurations de processeur et de profiler l'exécution d'une application donnée. Toute la phase d'exploration se situe à un haut niveau d'abstraction, ce qui permet d'accélérer cette phase. Suite à la phase d'exploration, le concepteur passe à la phase d'implémentation. Cette technologie génère automatiquement une description VHDL synthétisable. Cette description peut être synthétisée à l'aide des outils standards qu'il est possible de trouver sur le marché. En ce qui concerne la performance, l'aire et la consommation de puissance d'un processeur réalisé avec cette technologie, ceux-ci sont légèrement moins optimisés que pour un processeur réalisé manuellement. Cependant, le temps de conception se trouve fortement réduit.

2.4.2 Target Compiler

La technologie Target Compiler est un outil commercialisé pour accélérer le développement, la programmation et la vérification de noyau IP flexible sous forme de processeur embarqué et d'ASIC programmable. Comme pour la technologie Lisatek, cette technologie offre une flexibilité au niveau de l'architecture du processeur qu'il n'est pas possible de retrouver avec le Xtensa et le Nios. Cet environnement permet l'exploration architecturale à haut niveau, le développement d'outils logiciels et la génération automatique de la description matérielle du processeur en VHDL.

Avec cette technologie, le concepteur doit a priori définir l'architecture du processeur à l'aide du langage nML. Ce langage de haut niveau permet de définir l'architecture du processeur et son jeu d'instruction. Celui-ci permet d'effectuer de l'exploration architecturale rapide. Il est possible à l'aide de ce langage de supporter une vaste gamme de processeurs, du processeur à usage général à une architecture spécifique pour une application donnée comme les DSP. Ce langage permet de décrire le pipeline, la résolution d'aléas, un format variable d'encodage des instructions, ainsi que la définition d'instructions spécialisées.

Cette technologie repose principalement sur le « *Chess/Checker* » qui se trouve à être un ensemble d'outils pour la conception de processeur embarqué. L'outil « *Chess* » est en fait un compilateur C qui permet de traduire le programme de l'application dans un code machine (format Elf/Dwarf) optimisé pour le processeur visé. Pour sa part, l'outil « *Checkers* » est un simulateur de jeu d'instruction qui permet de simuler l'exécution d'un code machine avec une précision au niveau du cycle. Cet outil permet aussi la cosimulation avec d'autres modèles exécutables, comme des modèles HDL, SystemC et d'autres modèles ISS. Bien entendu, cette technologie offre d'autres outils pour l'assembleur/déssambleur, l'édition des liens, la génération de la description matérielle du processeur HDL et la génération de programme test. Une fois que l'exploration architecturale est terminée, il est possible de générer automatiquement la description HDL du processeur à partir de la description nML. Ce code synthétisable VHDL peut

être intégré avec d'autres modules et il peut être synthétisé à l'aide des outils standards du marché. De plus, il est spécifié qu'un certain nombre de scripts sont générés afin de faciliter la synthèse avec les outils les plus populaires.

2.4.3 Autres technologies

Il existe bien d'autres processeurs configurables sur le marché à part ceux qui ont été discutés précédemment. Outre ceux-ci, il existe sur le marché le processeur ARC Tangent de la Compagnie ARC [3][33]. Celui-ci est un processeur embarqué de type RISC pipeliné sur 4 étages. Il est possible d'intégrer certaines options prédéfinies au noyau du processeur, ainsi que des fonctionnalités DSP. Ce processeur vient sous forme RTL synthétisable et vient avec un ensemble d'outils logiciel.

La société Critical Blue [12][33] offre pour sa part un produit nommé Cascade. Ce produit est en fait un ensemble d'outils qui génère de façon automatique un coprocesseur spécifique à une application. La technique utilisée est de compiler le code de l'application, de repérer les boucles qui consomment le plus de temps d'exécution et d'en extraire un coprocesseur correspondant. Le coprocesseur est attaché au noyau du processeur par une interface bus et une mémoire partagée.

Pour leur part, Improv Systems [26][33] ont mis en marché un processeur nommé Jazz. Celui-ci est en fait un processeur DSP auquel il est possible d'introduire des instructions VLIW. La configuration de ce processeur est flexible et vise une application de traitement de signal. Ce processeur vient sous forme de code Verilog synthétisable, avec un environnement d'analyse temporelle et un environnement de vérification.

CHAPITRE 3

CONCEPTION D'INSTRUCTIONS SPÉCIALISÉES

La technologie des processeurs configurables permet d'ajouter des unités spécialisées au noyau du processeur afin d'accélérer l'exécution d'une application donnée. Cependant, il est important d'investiguer quelles sont les techniques qui permettent une amélioration. De plus, devant plusieurs instructions spécialisées, il faut être en mesure de déterminer quelles sont celles qui vont permettre d'atteindre les objectifs. Il est évident que cette technologie offre l'opportunité d'accélérer une application, par contre, il est nécessaire de déterminer les limites de celle-ci. Par conséquent, le présent chapitre est divisé en trois principales parties. La première est consacrée aux techniques d'optimisation, la seconde, aux métriques de qualité d'une instruction spécialisée, et la dernière, aux limites de l'accélération.

3.1 Techniques d'optimisation

Les processeurs configurables permettent d'obtenir des gains dans l'efficacité du système. Cependant, il est important de bien cibler la provenance de ces gains. Cette section décrit brièvement les techniques à utiliser afin d'obtenir des gains. Ces gains proviennent des registres spécialisés, de la fusion d'opérateurs, de la parallélisation des opérateurs, de la conception d'opérateurs vectoriels, du pipelinage des opérateurs et du partage des opérateurs fonctionnels.

3.1.1 Registres spécialisés

A priori, les processeurs configurables possèdent pour la plupart des registres spécialisés. Ces registres permettent d'emmagasiner des valeurs de façon temporaire afin d'effectuer

des opérations spécialisées sur ceux-ci. Ainsi, les variables redondantes peuvent être emmagasinées dans ces registres, ceci évite les instructions de chargement des données et les cycles de latence pour les accès mémoires.

La taille de ces registres peut être élevée, c'est-à-dire jusqu'à 128 bits par registre. Par conséquent, plusieurs valeurs juxtaposées peuvent prendre place dans les registres. Par exemple, il serait possible de placer côte à côte 4 variables de 16 bits dans un registre de 64 bits. Il n'existe pas de règles établies pour la conception de registres spécialisés. De plus, le nombre de registres n'est souvent pas limité. Bien entendu, la taille et le nombre de registres se reflètent dans la complexité matérielle.

L'usage des registres spécialisés est réservé avant tout pour les instructions spécialisées. En fait, il n'est pas très bénéfique d'utiliser des registres spécialisés sans unité fonctionnelle spécialisée. Effectivement, une opération conventionnelle effectuée sur un registre spécialisé nécessitera forcément un transfert de données du registre spécialisé vers les registres généraux. Ce transfert de données demandera bien entendu une instruction et un cycle d'exécution, ce qui se rapproche d'un chargement de données de la mémoire. Bref, le bénéfice découlant des registres spécialisés réside dans la présence des données proche des organes de calcul lors des opérations spécialisées.

3.1.2 Fusion d'instructions simples

Cette technique consiste à fusionner des instructions simples ensemble afin de former une nouvelle instruction [17]. L'utilisation de cette technique permet de réduire la taille du code et peut réduire l'utilisation des registres. En plus, la latence de la nouvelle instruction est souvent moins élevée que la latence des opérations simples combinées.

Pour certaines applications, il est plus facile de réaliser certaines fonctionnalités en matériel plutôt qu'en logiciel. Par exemple, une application qui nécessiterait de permuter les bits d'une donnée de 32 bits se réaliserait avec une longue séquence d'instructions. Cependant, cette même fonctionnalité peut être réalisée en un seul cycle en matériel. Ce

phénomène peut être illustré à l'aide de la figure 3.1. La version logicielle de la fonctionnalité de permutation de bits nécessitera forcément une instruction pour extraire le bit de départ (ANDI), une seconde instruction pour déplacer le bit à sa position finale (SHRI) et une dernière pour fusionner le bit au résultat final (ORI). Ainsi, la permutation nécessite 3 instructions par bits. Par contre, la version matérielle permet facilement de réaliser cette même fonctionnalité, il s'agit de permuter les fils des bits entre le registre d'entrée et celui de sortie. Il en résulte un gain de performance très important (3N où N est le nombre de bits à permuter) pour cette fonctionnalité et le matériel pour l'implémentation de cette instruction spécialisée est minime.

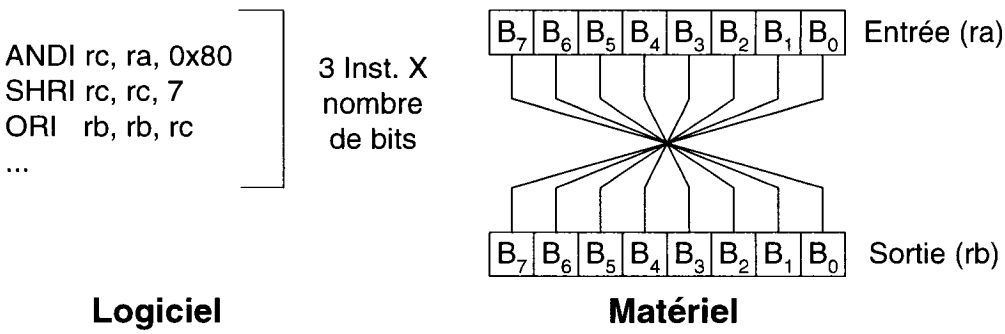

Figure 3.1 : Comparaison logiciel-matériel pour une permutation de bits

3.1.3 Opérations vectorielles

La technique des opérations vectorielles consiste à augmenter le débit des données en créant des unités fonctionnelles qui opèrent sur plusieurs éléments de donnée à la fois [17]. Les opérations vectorielles sont souvent référées comme SIMD (« *Single Instruction Multiple Data* »). Ces opérations sont caractérisées par les transformations qu'elles performent sur chaque élément de données et le nombre d'éléments qu'elles traitent en parallèle. L'un des avantages des opérations vectorielles est qu'elles évitent les instructions de surcharge pour le bouclage. De plus, un autre avantage des opérations vectorielles est qu'elles réduisent la taille du code. En fait, une seule instruction peut

effectuer une opération sur un ensemble de N éléments de données et de façon parallélisée. Les opérations vectorielles sont avantageuses pour les applications orientées flot de données, comme le traitement d'image et le traitement de signal.

Par exemple, une opération vectorielle d'addition sur deux vecteurs de 6 éléments chacun de 16 bits peut s'accomplir sur 3 cycles avec une unité fonctionnelle spécialisée. Par exemple, la figure 3.2 illustre comment les 6 éléments du vecteur A sont additionnés aux 6 éléments du vecteur B pour former le vecteur C. Le module matériel proposé comporte deux unités d'addition en parallèle qui produisent deux éléments de données du résultat par cycle. Si cette même fonctionnalité était réalisée en logiciel, il faudrait en premier lieu configurer les compteurs pour la boucle, additionner les éléments des vecteurs et recommencer jusqu'à la fin du vecteur. Il en résulterait des instructions additionnelles pour chaque itération de la boucle, ce qui est évité avec des opérations vectorielles.

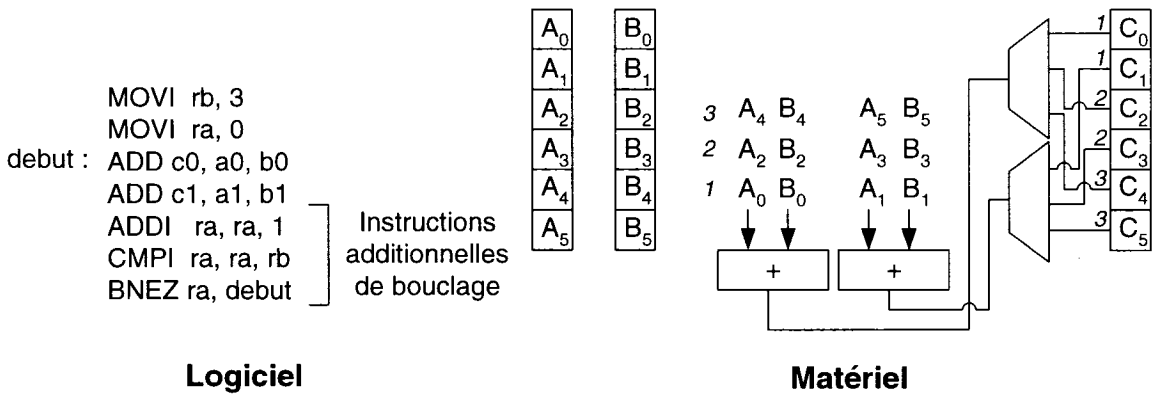

Figure 3.2 : Comparaison logiciel-matériel pour une opération vectorielle

3.1.4 Parallélisation sous forme VLIW

La technique de parallélisation sous forme VLIW (« *Very Long Instruction Word* ») permet à une instruction de contenir de multiples opérations indépendantes [17]. En fait, plusieurs opérations sont empaquetées en une seule, ce qui a pour conséquence d'augmenter les performances en termes du nombre d'instructions exécutées par cycle.

Une instruction VLIW est partitionnée en un certain nombre de champs, chacun des champs correspond à l'exécution d'une instruction. Ces opérations peuvent être de toute sorte comme des unités d'accès mémoire, de branchement, des unités logiques et arithmétiques ou même des modules spécialisés. L'empaquetage des instructions est déterminé lors de l'ordonnancement des instructions, i.e lors de la compilation.

Un processeur prenant avantage de ce type d'optimisation doit dans ces conditions avoir de multiples décodeurs en parallèle, soit l'un pour chaque instruction exécutée en parallèle. De plus, toutes les instructions pouvant être exécutées en parallèle doivent nécessairement avoir plusieurs unités logiques en parallèle, ce qui augmente considérablement le coût matériel du processeur par la réplication des unités logiques. À ces considérations, il faut ajouter que les opérations peuvent accéder aux mêmes registres, ce qui nécessite plusieurs ports de lecture pour chaque registre. À toutes ces implications, il faut mentionner que la taille du code est aussi augmentée, car l'ordonnancement des instructions ne permettra pas à chaque instruction de combler tous les champs de l'instruction VLIW. Ainsi, certaines instructions VLIW auront des champs libres ce qui sera traduit par du code inutile.

Par exemple, si un processeur contient trois unités fonctionnelles en parallèle, celui-ci aura la possibilité de réaliser trois opérations par cycle. Si les opérations supportées par ces unités fonctionnelles sont les opérations sur des entiers, la multiplication et les accès mémoires, l'instruction contiendra par conséquent trois champs, soit un pour chaque unité fonctionnelle. Les registres devront avoir autant de ports de lecture qu'il est requis pour supporter des accès multiples à un même registre lors de l'exécution. La figure 3.3 illustre ce type d'optimisation.

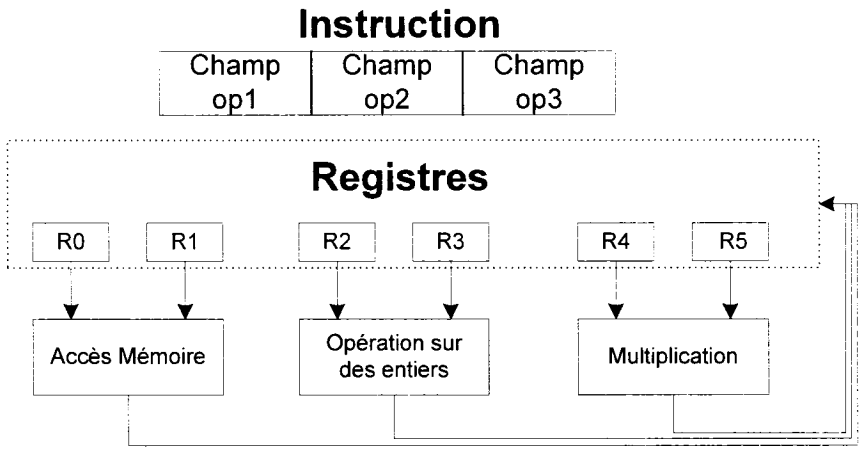

Figure 3.3 : Parallélisation sous forme VLIW

3.1.5 Instructions Pipelinées

La création d'instruction spécialisée peut donner naissance à de longs chemins critiques. Afin de ne pas trop réduire la fréquence d'horloge du processeur, il est important de créer un pipeline sur l'instruction spécialisée. La technique consiste à scinder le chemin critique en un certain nombre d'étages. L'exécution de l'instruction spécialisée introduira des données dans le premier étage du pipeline et le résultat sera obtenu à la sortie du dernier étage. Chaque étage entre le premier et le dernier aura comme entrées les sorties de l'étage précédent. Le pipelinage aura pour conséquence d'augmenter le débit de production des résultats tout en conservant la latence essentiellement inchangée (temps entre la production des premiers résultats et les premières entrées). Si une instruction spécialisée est scindée en N étages, elle produit par conséquent le premier ensemble de résultats avec N exécutions de l'instruction et chaque appel subséquent produit un nouvel ensemble de résultats.

Par exemple, si une division de deux nombres en virgule fixe prend 8 cycles à s'exécuter séquentiellement, celle-ci peut être facilement pipelinée. Comme la figure 3.4 le démontre, cette division peut être morcelée en 4 étapes distinctes. Au lieu d'être exécuté de façon séquentielle, c'est-à-dire d'introduire des données et d'attendre le résultat

correspondant, celle-ci peut être exécuté de façon parallèle. Cela aura pour conséquence de réduire le temps d'exécution. S'il faut réaliser 4 divisions consécutives, il faudra 32 cycles d'exécution de façon séquentielle. Cependant, de façon pipelinée, il faudra 14 cycles pour exécuter ces 4 divisions consécutives (4x2 +1x6, où les 6 cycles représentent la latence pour remplir le pipeline).

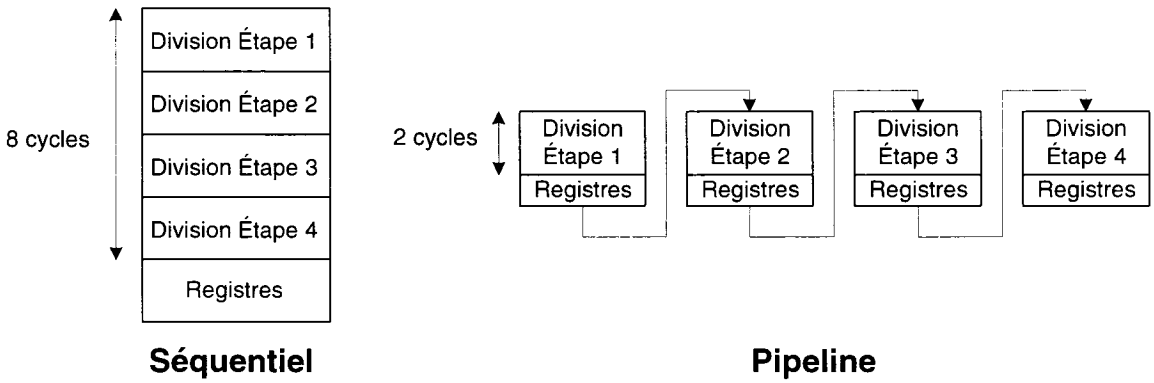

Figure 3.4 : Instruction spécialisée pipelinée

3.1.6 Partage des unités opératives

Les opérations des unités spécialisées peuvent être redondantes. En d'autres mots, deux instructions spécialisées peuvent contenir la même opération complexe, cette opération peut être conséquemment partagée entre les instructions. La technique proposée consiste donc à ajouter des multiplexeurs à l'entrée d'un opérateur complexe redondant et un démultiplexeur à sa sortie. Ainsi, les entrées et sorties de l'opérateur complexe sont sélectionnées en fonction de l'instruction à exécuter.

Par exemple, deux instructions spécialisées pourraient contenir chacune des multiplieurs. Si les multiplications sont effectuées sur certains registres pour une instruction spécialisée et d'autres registres pour la seconde, les multiplieurs peuvent être partagés. Dans ce cas, le matériel ajouté se résume à de simples multiplexeurs pour les entrées et des démultiplexeurs pour les sorties des multiplieurs.

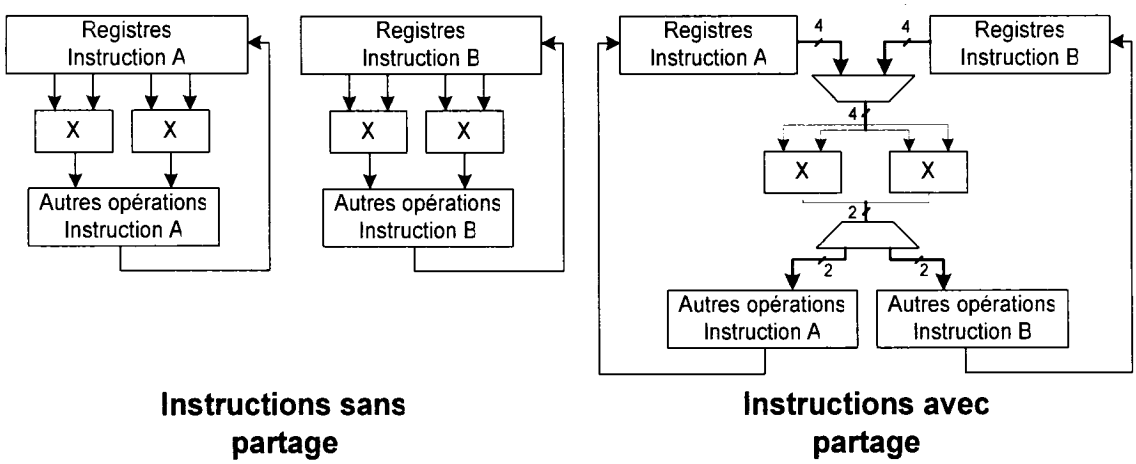

Figure 3.5 : Partage des unités fonctionnelles

Cette technique a l'avantage de réduire la complexité matérielle des instructions spécialisées. Cependant, il ne faut pas partager des opérateurs trop simples, car la complexité matérielle ajoutée par les multiplexeurs sera plus élevée que la réduction due au partage. Lorsqu'un doute persiste au niveau de la pertinence de partager un opérateur, il faut effectuer une étude de complexité matérielle pour les deux configurations.

Les techniques d'optimisation discutées permettent d'obtenir des gains en performance sur l'exécution d'une application donnée tout en limitant les coûts du matériel additionnel requis pour y arriver. L'une des contributions à ce gain vient du fait que le passage des données entre les instructions est éliminé en combinant plusieurs instructions conventionnelles en une seule instruction spécialisée. De plus, si une seule instruction spécialisée remplace plusieurs instructions conventionnelles, il en résulte une réduction de la taille du code pour une même fonctionnalité. Cette réduction du code a pour conséquence de réduire le nombre d'instructions chargées en mémoire, ce qui peut contribuer à réduire les ratés de l'antémémoire (« *cache misses* »).

3.2 Métriques d'efficacité

Plusieurs instructions spécialisées peuvent être générées pour une même application. Chaque instruction peut être aussi une variante d'une autre. Il est par conséquent important de pouvoir mesurer la pertinence de chacune des instructions. Pour ce faire, il faut introduire des métriques d'efficacité qui permettront d'effectuer une sélection parmi les instructions spécialisées. Les métriques d'efficacité pour les instructions spécialisées sont les gains en performance, les performances en fonction de la complexité matérielle ajoutée et la réduction de la consommation de puissance. La description des métriques est effectuée en premier lieu pour l'introduction d'une seule instruction spécialisée. Ensuite, une section démontre comment calculer une métrique pour un ensemble de M instructions spécialisées.

3.2.1 Gain en performance

La conception d'instructions spécialisées est souvent réalisée afin d'obtenir un gain sur le nombre de cycles d'exécution d'une application. L'introduction d'une instruction spécialisée peut par conséquent remplacer un ensemble d'instructions conventionnelles. Ceci se traduit par un gain en performance sur l'application totale. Dans le gain en performance, il faut a priori prendre en compte plusieurs facteurs. L'un de ces facteurs est la fréquence d'horloge avant et après l'ajout d'instructions spécialisées. Un autre de ces facteurs est le nombre d'instructions conventionnelles que cette nouvelle instruction remplace, qui se traduit par un gain en nombres de cycles. De plus, il faut tenir compte de la redondance de l'instruction dans l'application donnée. Effectivement, plus un ensemble d'instructions se retrouve régulièrement dans l'application, plus le gain en performance sera important suite à l'ajout de l'instruction spécialisée.

En premier lieu, la fréquence d'horloge sera considérée constante avant et après l'introduction de la nouvelle instruction. Ceci est réalisable en concevant la nouvelle

instruction sur plusieurs étages de pipeline. Afin de déterminer le nombre d'étages nécessaires, il suffit d'utiliser l'équation suivante.

$$\text{Nombre d'étages du pipeline} \geq \left\lceil \frac{\text{Chemin critique (ns)}}{\text{Période d'horloge (ns)}} \right\rceil \quad \textbf{(3-1)}$$

En d'autres mots, il suffit de scinder le chemin critique de l'instruction spécialisée dans un nombre d'étages suffisant pour que la fréquence d'horloge soit conservée. Cette façon de faire permet d'éviter de ralentir le reste de l'application pour en accélérer une partie. En fait, il est généralement possible d'éviter que le chemin critique du processeur ne soit dans l'instruction spécialisée.

En conservant la fréquence d'horloge constante, le gain en performance de l'instruction spécialisée i peut être déterminé par le rapport du nombre de cycles d'exécution avant (N) et après l'ajout de cette instruction ($N-\gamma_i\alpha_i$). Par conséquent, le gain en nombre de cycles G_i est fonction du nombre d'instructions conventionnelles éliminées α_i et du nombre de répétitions γ_i de l'ensemble d'instructions remplacées dans l'application.

$$G_i = f(\gamma_i, \alpha_i) = \frac{\text{Nb de cycles d'exécution sans Inst.}}{\text{Nb de cycles d'exécution avec Inst.}} = \frac{N}{N - \gamma_i\alpha_i} = \frac{1}{1 - \frac{\gamma_i\alpha_i}{N}} \quad \textbf{(3-2)}$$

Après l'ajout de l'instruction i, l'exécution de l'application prend $\gamma_i\alpha_i$ cycles de moins. Ainsi, plus ce nombre de cycles est grand, plus le gain en nombre de cycles G_i est important. Deux métriques de performance possibles sont le gain en nombre de cycles G_i comme formulé par l'équation 3-2 ou tout simplement le nombre de cycles en moins ($\gamma_i\alpha_i$) engendré par l'instruction spécialisée.

Le résultat des performances est parfois mesuré en fonction d'une section du code qui est accélérée. Ainsi, si une portion λ du programme est accélérée par un facteur β par l'instruction spécialisée i, le gain en nombre de cycles peut être aussi calculé avec l'équation 3-3. Cette équation est une variante de la loi d'Amdhal, cette loi sera définie dans une prochaine section.

$$G_i = f(\lambda_i, \beta_i) = \frac{1}{\text{Portion non accélérée} + \frac{\text{Portion accélérée}}{\text{Facteur d'accélération}}} = \frac{1}{(1-\lambda_i) + \frac{\lambda_i}{\beta_i}} \quad \textbf{(3-3)}$$

Il est possible de faire le lien entre l'équation 3-2 et 3-3 de la façon suivante (tel qu'exprimé par l'équation 3-4) : portion soustraite = portion accélérable – portion résultante après accélération.

$$\frac{\alpha_i \gamma_i}{N} = \lambda_i - \frac{\lambda_i}{\beta_i} \quad \textbf{(3-4)}$$

Les dernières équations calculent le gain en nombre de cycles et non le gain en performance totale. En d'autres mots, elles ne tiennent pas compte du changement de la fréquence d'horloge. Le gain en nombre de cycles est équivalent au gain en performance seulement si la fréquence d'horloge reste constante. Ainsi, le gain en performance P_i peut être calculé avec l'équation 3-5.

$$P_i = \frac{\text{Temps d'exécution sans Inst.}}{\text{Temps d'exécution avec Inst.}} = G_i \frac{T_{base}}{T_i} = G_i \frac{f_i}{f_{base}} \quad \textbf{(3-5)}$$

T_{base}, f_{base} : Période, fréquence de l'horloge de la configuration de base du processeur

T_i, f_i : Période, fréquence de l'horloge après l'ajout d'instruction spécialisée i

Avec cette dernière équation, il faut éliminer toutes les instructions spécialisées dont le rapport du nombre de cycles d'exécution est moins élevé que le rapport des fréquences. Si un tel cas se produit, il n'y a aucun gain en performance, mais plutôt une dégradation. Ceci est exprimé par l'équation 3-6.

$$P_i = G_i \frac{f_i}{f_{Base}} > 1 \Rightarrow G_i = \frac{1}{(1-\lambda_i) + \frac{\lambda_i}{\beta_i}} > \frac{f_{Base}}{f_i} \quad \textbf{(3-6)}$$

3.2.2 Performances en fonction de la complexité matérielle ajoutée

Un coût matériel est associé à chacune des instructions spécialisées. Il est possible de refléter ce coût dans une métrique qui tient compte à la fois des performances et du coût matériel. Le coût matériel peut être mesuré en nombre de portes ajoutées pour une instruction spécialisée ou par l'aire que celle-ci utiliserait, l'aire étant considérée comme une fonction du nombre de portes logiques. Cette métrique doit démontrer que le gain en performance est plus élevé que le coût en matériel. Ainsi, une instruction spécialisée sera préférée à une autre si son gain en performance est plus élevé pour un même coût matériel ou si le coût matériel est moins élevé pour un même gain en performance. À priori, il faut définir le coût matériel relatif C de l'instruction spécialisée i.

$$C_i = \frac{A_{Base} + A_i}{A_{Base}} = 1 + \frac{A_i}{A_{Base}} \tag{3-7}$$

A_{Base} : Aire du processeur de base auquel une instruction spécialisée a été ajoutée

A_i : Aire ajoutée par l'instruction spécialisée i

Ce coût matériel relatif est mesuré par rapport à la configuration de base du processeur. Il démontre quel pourcentage de l'aire de base représente le processeur avec l'instruction spécialisée. Avec ce coût matériel relatif, il est possible de concevoir une métrique performance-aire PA_i pour l'instruction spécialisée i :

$$PA_i = \frac{P_i}{C_i} = \frac{G_i \frac{T_{Base}}{T_i}}{\left(1 + \frac{A_i}{A_{Base}}\right)} = \frac{f_i / f_{Base}}{\left(\left(1 - \lambda_i\right) + \frac{\lambda_i}{\beta_i}\right)\left(1 + \frac{A_i}{A_{Base}}\right)} \tag{3-8}$$

Cette métrique pondère le gain des performances par le coût du matériel ajouté : plus cette métrique est élevée, plus les gains en performance sont importants par rapport à l'ajout de matériel. Par exemple, si une instruction spécialisée augmente les performances par un facteur 2 et que l'ajout du matériel double l'aire du processeur, la métrique PA résultante sera de 1. Une métrique PA de 1 dénote plutôt d'un compromis entre l'aire et les performances que d'un réel bénéfice en efficacité de calcul.

Il est à noter que cette dernière métrique est équivalente à un rapport de produits AT entre la configuration de base du processeur et celle avec une instruction spécialisée. Cette équivalence est démontrée dans l'équation 3-9. Cette façon de considérer la métrique sera utilisée plus tard lors de l'analyse des résultats.

$$PA_i = \frac{P_i}{C_i} = \frac{\begin{matrix}T \text{ exécution sans inst. i} \\ \hline T \text{ exécution avec inst. i}\end{matrix}}{\begin{matrix}\text{A processeur avec inst. i} \\ \text{A processeur sans inst. i}\end{matrix}} = \frac{AT \text{ sans inst. i}}{AT \text{ avec inst. i}} = Rapport\ AT_i \quad \textbf{(3-9)}$$

3.2.3 Réduction de la consommation de puissance

Avec les processeurs configurables, il est aussi possible de réduire la consommation d'énergie. Cela est réalisable notamment en considérant un objectif de performance fixe et facilement atteignable. Avec cette performance visée, il s'agit ensuite d'ajouter des instructions spécialisées pour augmenter les performances par un facteur plus élevé que celui de la puissance. Ainsi, si les performances sont augmentées par un facteur 4 et que la consommation de puissance est augmentée par un facteur 2, il en résulte une diminution de la consommation d'énergie par un facteur 2 pour les mêmes performances de calcul. Par conséquent, le processeur sera actif une fraction du temps, correspondant au ratio des performances. Une métrique d'efficacité qui reflète ce bénéfice sera donc un rapport entre le gain des performances de calcul et l'ajout de consommation de puissance.

Avant de poursuivre avec la définition de la métrique, il faut a priori définir le coût en consommation de puissance avec l'équation 3-10.

$$CPW_i = \frac{PW_i}{PW_{Base}} \quad \textbf{(3-10)}$$

PW_{Base} : Consommation de puissance de la configuration de base du processeur

PW_i : Consommation de puissance du processeur avec l'instruction spécialisée i

CPW_i : Consommation de puissance relative pour l'ajout de l'instruction spécialisée i

Une métrique performance-puissance (PP_i) pour l'instruction spécialisée i peut être définie à l'aide de l'équation 3-11.

$$PP_i = \frac{P_i}{CPW_i} = \frac{G_i \dfrac{f_i}{f_{Base}}}{\dfrac{PW_i}{PW_{Base}}} = \frac{PW_{Base}}{PW_i}\left(\frac{1}{(1-\lambda_i)+\dfrac{\lambda_i}{\beta_i}}\right)\frac{f_i}{f_{Base}} \qquad \textbf{(3-11)}$$

Cette métrique démontre que plus le gain en performance de calcul sera élevé par rapport à la consommation de puissance relative, plus l'instruction spécialisée i contribue à la réduction de puissance. Il est à observer que cette métrique est en fait un rapport entre l'énergie consommée par le processeur de base et celle consommée par le processeur avec l'instruction spécialisée i pour effectuer un même calcul. Ce rapport d'énergie est établi pour une même performance de calcul. Il est possible de constater que si la consommation de puissance est considérée constante sur un intervalle de temps, l'énergie consommée équivaut au produit de la puissance et du temps, soit l'aire sous la courbe puissance-temps de la figure 3.6.

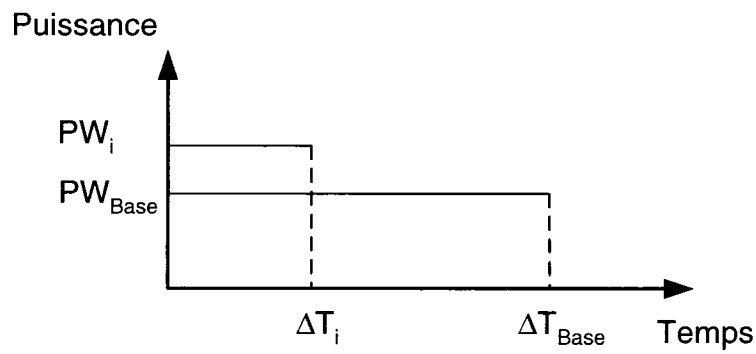

Figure 3.6 : Réduction de puissance avec l'instruction spécialisée i

Sur ce dernier graphique, l'intervalle de temps ΔT représente le temps requis pour effectuer une tâche spécifique. Ainsi, l'aire sous la courbe formée par $PW_{Base}\Delta T_{Base}$ représente l'énergie consommée par le processeur de base et l'aire sous la courbe formé

par $PW_i\Delta T_i$ représente l'énergie consommée par le processeur avec l'instruction spécialisée i. Dans ce cas, l'intervalle de temps est défini par le produit du nombre de cycles N et de la période de l'horloge T_{Base}. Avec l'instruction spécialisée i, le nombre de cycles passe de N à $(N-\gamma_i\alpha_i)$ et la période d'horloge passe de T_{base} à T_i.

$$PP_i = \frac{E_{Base}}{E_i} = \frac{PW_{Base}\Delta T_{Base}}{PW_i\Delta T_i} = \frac{PW_{Base}NT_{Base}}{PW_i(N-\gamma_i\alpha_i)T_i} = \frac{PW_{Base}}{PW_i}\left(\frac{N}{N-\gamma_i\alpha_i}\right)\frac{f_i}{f_{Base}} \quad \textbf{(3-12)}$$

Cette métrique PP doit être interprétée comme étant le facteur par lequel l'énergie consommée a été diminuée. Par conséquent, plus cette métrique est élevée, meilleure est la réduction d'énergie consommée que permet l'instruction i. Une métrique plus petite que 1 signifie une augmentation de l'énergie consommée plutôt qu'une réduction.

3.2.4 Calcul de métrique pour M instructions spécialisées

Le calcul d'une métrique de performance pour M instructions spécialisées est effectué par la somme des gains de chacune des instructions. Il est important que chaque instruction spécialisée prise dans l'ensemble soit indépendante des autres, c'est-à-dire que l'utilisation d'une instruction spécialisée n'influence pas la fonctionnalité de l'autre. En d'autres mots, chaque instruction spécialisée doit accélérer une section distincte de l'application. De plus, il faut prendre en considération que la fréquence d'horloge du processeur spécialisée sera la fréquence d'horloge minimale imposée par la plus lente des M instructions spécialisées. Ainsi, la fréquence d'horloge est définie à l'aide de l'équation 3-13.

$$f_M = \min(\forall_{i=1}^{M} f_i) \quad \textbf{(3-13)}$$

Ceci signifie que la fréquence d'horloge du processeur spécialisé f_m sera imposée par le plus long chemin critique des M instructions. Si le chemin critique ne se retrouve pas parmi les M instructions spécialisées, la fréquence d'horloge est la même que celle de la configuration de base et le facteur f_M / f_{Base} peut être ignoré dans le calcul du gain des

performances. Le gain des performances P_M pour l'introduction de M instructions spécialisées peut être calculé de la façon suivante :

$$P_M = f(\lambda_i, \beta_i) = \frac{1}{\left(1 - \sum_{i=1}^{M} \lambda_i\right) + \sum_{i=1}^{M} \frac{\lambda_i}{\beta_i}} \left(\frac{f_M}{f_{Base}}\right) \quad \textbf{(3-14)}$$

En ce qui concerne la métrique performance-aire PA_M pour M instructions, il faut additionner le bénéfice de chacune des instructions pour le gain des performances et l'aire de toutes les instructions spécialisées dans le coût matériel relatif. Pour le calcul du gain des performances, les équations précédentes s'appliquent. Pour ce qui a trait à la complexité matérielle relative, il faut additionner l'aire de chaque instruction spécialisée pour connaître la complexité matérielle ajoutée. Ainsi, le coût matériel pour M instructions spécialisées C_M peut se calculer à l'aide de l'équation suivante.

$$C_M = \frac{A_{Base} + \sum_{i=1}^{M} A_i}{A_{Base}} = 1 + \sum_{i=1}^{M} \frac{A_i}{A_{Base}} \quad \textbf{(3-15)}$$

Comme mentionné précédemment, il est possible de partager des unités opératives entre les instructions spécialisées. Par conséquent, il faut tenir compte de ce partage de complexité matérielle dans le calcul. Pour ce faire, il s'agit de calculer l'aire totale des deux instructions spécialisées comme si elles ne constituaient qu'une seule instruction et d'additionner les bénéfices pour le gain en performances.

$$PA_M = \frac{P_M}{C_M} = \frac{G_M \frac{f_M}{f_{Base}}}{\left(1 + \sum_{i=1}^{M} \frac{A_i}{A_{Base}}\right)} \quad \textbf{(3-16)}$$

La métrique de réduction de puissance se calcule de la même façon que les précédentes, c'est-à-dire que les équations du gain des performances s'appliquent et qu'il ne reste qu'à trouver la consommation de puissance relative. Cette consommation de puissance relative se calcule avec la puissance totale du processeur spécialisé avec les M instructions.

$$CPW_M = \frac{PW_M}{PW_{Base}} \tag{3-17}$$

$$PP_M = \frac{P_M}{CPW_M} = \frac{G_M \frac{f_M}{f_{Base}}}{\frac{\mathrm{PW_M}}{\mathrm{PW_{Base}}}} = \frac{PW_{Base}}{PW_M} G_M \frac{f_i}{f_{Base}} \tag{3-18}$$

3.3 Limites de l'accélération

Les résultats obtenus confirment qu'un gain considérable en performance peut être obtenu à l'aide d'instructions spécialisées. Bien qu'une accélération dans l'exécution d'une application puisse être obtenue, il est important de déterminer quelles sont les limites à une telle accélération. La présente section s'attarde à cette question, en commençant par la loi d'Amdhal, des contraintes physiques, des limites du chargement des données et des limites de l'accélération par parallélisation des opérateurs.

3.3.1 Loi d'Amdhal

Cette loi stipule qu'il est possible de diviser toute application en une portion accélérable et une portion non-accélérable. Cette loi dicte que l'amélioration des performances en utilisant une portion accélérée est limitée à la fraction du temps que cette portion accélérée est utilisée. La loi d'Amdhal donne un aperçu rapide de l'accélération qu'il est possible d'atteindre avec quelques améliorations, laquelle dépend de deux facteurs :

- La fraction du temps de calcul du code original (α) qui peut être converti pour tirer profit des améliorations. Ainsi, si une application prend 50 secondes à s'exécuter et qu'une section de 30 secondes peut être accélérée, la fraction accélérable est de 30/50. Cette fraction accélérable est toujours plus petite ou égale à 1.

- Le facteur d'accélération appliqué sur la partie accélérable (s). Ce facteur d'accélération est en fait le temps d'exécution original sur le temps d'exécution avec les améliorations apportées. Ainsi, si la section à accélérer est de 30 secondes et qu'après les améliorations, cette section prend 15 secondes à s'exécuter, le facteur d'accélération est donc de 2 (30/15=2).

La loi d'Amdhal se traduit mathématiquement par l'équation 3-19. L'équation de gauche s'applique pour une seule portion accélérable d'un facteur s. Tandis que l'équation de droite s'applique pour N sections accélérables, chacune ayant un facteur d'accélération s_i. En fait, une application réelle est habituellement divisée en plusieurs sections. Chacune des sections est prise individuellement et certaines améliorations sont portées à chacune d'elle, donc cette technique produit des facteurs d'accélération différents pour chacune d'elles.

$$G = \frac{1}{(1-\alpha)+\frac{\alpha}{s}} \; ; \; G = \frac{1}{\left(1-\sum_{i=1}^{N}\alpha_i\right)+\sum_{i=1}^{N}\frac{\alpha_i}{s_i}}; \qquad \textbf{(3-19)}$$

G : Gain total sur l'exécution de l'application
α : portion de l'application accélérable
s : facteur d'accélération de la portion accélérable

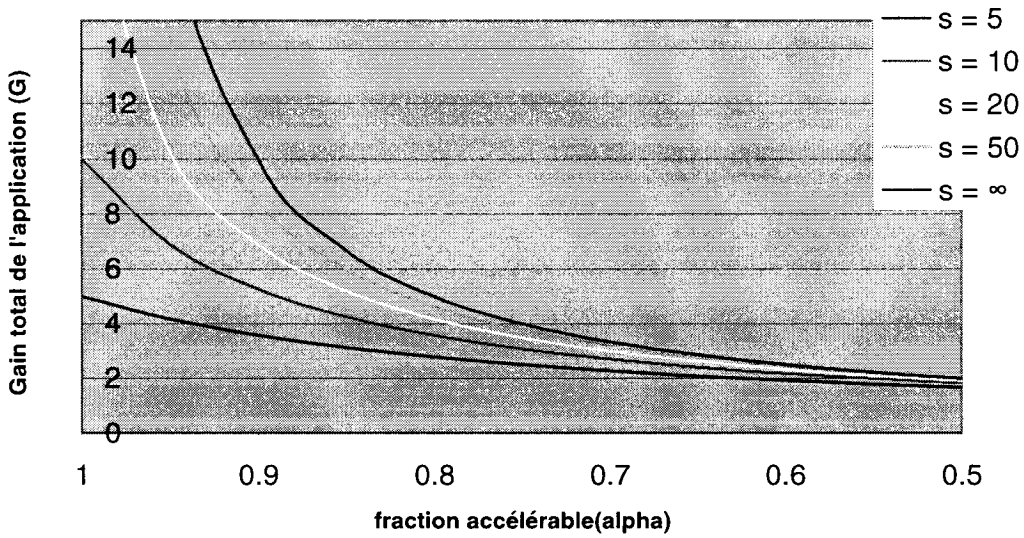

Figure 3.7 : Gain total en fonction de la portion accélérable

La figure 3.7 illustre l'évolution du gain total (G) de l'application en fonction de la portion accélérable. Plusieurs facteurs d'accélération ont été utilisés, soit 5, 10, 20, 50 et infini. La courbe du facteur d'accélération infini représente la limite théorique du gain si le facteur d'accélération pouvait atteindre une valeur infiniment grande. En d'autres mots, cette courbe illustre le gain atteignable si la portion accélérable avait un temps d'exécution négligeable par rapport à celui de la portion non-accélérable. Ce qui ressort de ce graphique, c'est qu'il n'est pas possible d'obtenir de très grands gains lorsque la portion accélérable est plus petite que 90%. La portion accélérable de l'application est en fait une limite de l'accélération atteignable avec des instructions spécialisées. Par conséquent, avant de considérer accélérer une application, il faut s'assurer qu'une grande portion de l'application soit accélérable. Donc, comme règle du pouce, pour obtenir des facteurs d'accélération élevés, il faut que l'ensemble des parties considérées pour l'accélération totalise plus de 90% du temps d'exécution de l'application.

3.3.2 Contraintes

Comme mentionné dans la section précédente, la portion accélérable de l'application est une limite à l'accélération. Cependant, il existe d'autres limites à l'accélération. Plusieurs de ces limites sont imposées par la technologie du processeur configurable utilisé. Parmi ces contraintes imposées, voici les plus courantes :

- Nombre d'opérandes d'une instruction (entrées et sorties)
- Aire attribuée à l'unité spécialisée
- Nombre d'instructions spécialisées permises
- Contrôle au sein de l'application
- Bande passante de la mémoire

Le nombre d'opérandes fournis à une instruction est souvent limité à 2 entrées et une sortie. C'est le cas pour la technologie Xtensa et le Nios. Cette limite en terme d'interface avec le noyau peut limiter l'impact des instructions spécialisées, c'est-à-dire le facteur d'accélération atteignable avec celles-ci. Cependant, cette limite s'applique surtout pour l'interface entre les instructions spécialisées et les registres généraux du processeur. Cette limite est imposée par le nombre de ports d'écriture et de lecture de ces registres. Ainsi, il est possible d'augmenter le nombre d'opérandes pour les instructions spécialisées en intégrant des registres spécialisés au noyau. Cependant, cette façon de faire nécessite un coût en termes de temps d'exécution relatif au chargement des registres si les paramètres ne sont pas déjà contenus dans ces registres. En somme, cette technique est intéressante lorsque les valeurs sont déjà dans les registres ou que la réduction du nombre de cycles d'exécution relatif à l'augmentation du nombre d'opérandes dépasse largement le coût de chargement des paramètres.

À cette dernière contrainte, il faut considérer aussi la contrainte physique en termes d'aire. Il est possible d'intégrer plusieurs instructions spécialisées au noyau d'un processeur configurable. Cependant, l'aire attribuable aux instructions spécialisées peut être un facteur limitatif. Bien entendu, cette aire n'est pas infinie, il faut considérer une

aire raisonnable pour l'unité spécialisée. Avec une plate-forme FPGA, le nombre de cellules logiques (ou « *Logic Cells* », *LC*) est limité physiquement à un certain nombre (relatif au type de FPGA), il n'est pas possible d'excéder cette limite à part en changeant le FPGA par un plus gros. De plus, avec ce type de plate-forme, il faut considérer que lorsque l'aire de l'unité spécialisée augmente, les délais de propagation dus au routage des signaux augmentent. Cette augmentation de délai réduit conséquemment la fréquence d'opération du processeur, ce qui affecte l'accélération. En ce qui concerne les plates-formes ASIC, il faut considérer une aire attribuable à l'unité spécialisée réalisable sur un dé de silicium. De plus, il ne faut pas perdre de vue qu'il faut aussi considérer l'aire du noyau du processeur, les bus de communication et la mémoire. De plus, dans le cadre des FPGA, il faut prévoir que si la complexité matérielle de l'unité spécialisée devient trop élevée, il risque d'y avoir des délais importants pour la propagation des signaux. À moins d'une forte restriction sur l'aire, ce facteur n'est cependant pas le plus limitatif sur l'accélération.

Un autre facteur limitatif à l'accélération est le nombre d'instructions spécialisées qu'il est possible d'implémenter avec une technologie. Par exemple, le noyau du Nios permet l'implémentation de seulement 5 instructions spécialisées. Pour sa part, la technologie Xtensa permet d'implémenter au moins 32 instructions spécialisées. Avec cette technologie, il est possible d'utiliser d'autres « opcodes » que ceux réservés à cet effet pour les instructions spécialisées avec la version T1050.0. Par contre, si d'autres « opcodes » sont utilisés, Tensilica ne garantit aucunement que l'unité spécialisée soit compatible avec les versions ultérieures. Ce nombre limité d'instructions spécialisées provient du nombre d'« opcodes » possible d'ajouter au décodeur. Par contre, cette contrainte peut être surmontée en prenant comme paramètre dans l'instruction un registre qui spécifie le type de fonctionnalité. Il va de soi que cette façon de faire ne peut être utilisé sans payer le coût du chargement de ce registre. Ainsi, il faut prendre en considération le cycle de chargement de ce registre pour l'exécution de l'instruction. Si cette technique est empruntée, il est recommandé de n'utiliser qu'une seule instruction

spécialisée et d'encoder les instructions spécialisées les moins souvent utilisées. Ceci aura pour effet de ne pas ralentir les instructions spécialisées les plus courantes, donc cela réduit l'impact sur l'accélération.

En ce qui concerne le contrôle au sein d'une application, il faut mentionner que les instructions de branchement ne peuvent pas être introduites au sein d'instructions spécialisées. Ainsi, ces instructions de branchements délimitent les régions de code pour lesquelles il est possible d'introduire des instructions spécialisées, ce qui se résume à une contrainte sur l'accélération. Pour se défaire de cette limitation, il est possible de modifier le code afin d'augmenter la grosseur des régions. Le lecteur intéressé à connaître davantage sur le sujet est invité à consulter l'annexe D (D.6, 7, 9, 11).

Si l'application est orientée « donnée », c'est-à-dire qu'un grand nombre de données circulent entre le processeur et la mémoire, la bande passante peut facilement devenir un goulot d'étranglement. Cependant, avec certaines technologies, l'interface mémoire peut être modifiée en augmentant la largeur des accès mémoires. Par exemple, il est possible avec la technologie Xtensa d'augmenter la largeur des accès aux données jusqu'à 128 bits (32, 64 et 128). Par contre, ce type de modification ne peut pas être réalisé avec le processeur Nios. Si une instruction spécialisée nécessite un grand nombre de données, il y aura un coût en nombre de cycles requis pour le chargement des données qui ne se retrouvent pas dans les registres. Ce coût peut être réduit en augmentant les accès mémoires. En somme, le coût relatif au chargement des données est diminué de moitié à chaque fois que la bande passante est doublée.

3.3.3 Chargement de données

Le chargement de données lors de l'exécution d'instructions spécialisées peut devenir une limite à l'accélération. Celle-ci devient plus contraignante quand le temps de chargement de données devient considérable par rapport au temps d'exécution de l'instruction spécialisée. Par chargement de données, il est sous-entendu le nombre de

cycles nécessaires pour que les variables inhérentes à une instruction se retrouvent dans les registres spécifiques (entrées de l'instruction). Pour étudier l'impact du chargement de données sur le gain local, il faut en premier lieu définir le gain local (s) comme étant le rapport entre le temps d'exécution original et le temps d'exécution avec l'unité spécialisée. Dans cette analyse, il est pris pour hypothèse que la fréquence d'horloge reste la même avant et après l'introduction d'instructions spécialisées. Par conséquent, le gain local se résume à un rapport du nombre de cycles d'exécution. Considérons qu'originalement la section prend N cycles pour s'exécuter. Après optimisation, il faut considérer que le nombre de cycles pour le chargement de données est n_i et que le nombre de cycles d'exécution de l'instruction est n_e (une fois les registres chargés). Suite à ces considérations, il est possible de définir le gain local effectif ($1/\lambda$). Ce gain est celui obtenu si le nombre de cycles de chargement était nul. Les équations suivantes résument ce raisonnement.

$$s = \frac{N}{n_i + n_e} \quad \text{si } \lambda = \frac{n_e}{N}, \beta = \frac{n_i}{n_e} \rightarrow s = \frac{1}{(\beta+1)\lambda} \tag{3-20}$$

N : Nombre de cycles d'exécution original

n_e : Nombre de cycle d'exécution de l'instruction spécialisée

n_i : Nombre de cycle pour le chargement des données

À présent, si le gain local en pourcentage du gain effectif est présenté en fonction de la fraction de l'exécution pour le chargement des données, cela donne le résultat présenté à la figure 3.8. Cette figure montre que le gain local décroît rapidement si le temps pour le chargement des données n'est pas une infime fraction du temps d'exécution. Par conséquent, il faut que les variables d'une instruction spécialisée soient déjà dans les registres pour que le matériel réalise le maximum de son gain effectif (β=0). Cela se résume à ajouter des registres spécialisés pour que les variables soient déjà contenues dans ceux-ci. Si la quantité de variables est trop élevée pour leur dédier chacune un registre, il est recommandé d'augmenter la bande passante mémoire pour réduire le temps

de chargement. En somme, plus le temps de chargement est faible, plus le matériel est utilisé à sa pleine efficacité. De plus, il peut être plus valable dans certains cas d'augmenter la bande passante de la mémoire que d'ajouter du matériel pour une unité spécialisée.

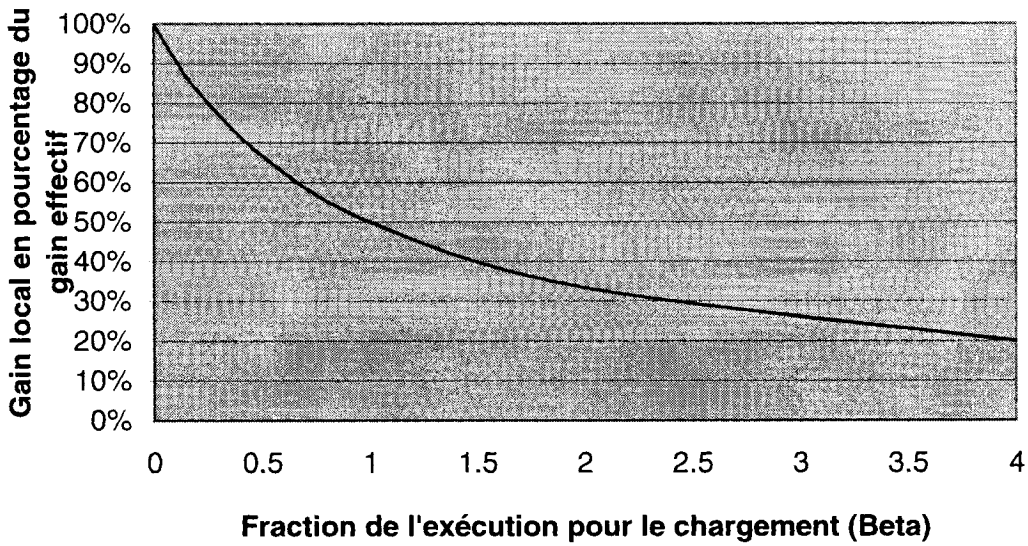

Figure 3.8 : Gain local en pourcentage du gain effectif en fonction de la fraction de l'exécution pour le chargement

3.3.4 Parallélisation des opérations dans un pipeline

Pour certains traitements, il est avantageux d'effectuer plusieurs opérations en parallèle avec une instruction spécialisée. Cependant, il est important d'étudier l'évolution de l'accélération locale par rapport à l'augmentation du parallélisme. Dans cette analyse, les opérations sont considérées comme étant pipelinées sur n étages. Par conséquent, il faut n cycles de traitement entre l'entrée des données et la sortie correspondante. Dans le premier cas, les données sont injectées à chaque coup d'horloge avant même que le traitement précédent ne soit terminé, i.e. que les entrées sont envoyées consécutivement. Il est possible d'illustrer la situation à l'aide de la figure 3.9. S'il faut effectuer N opérations similaires avec un pipeline de n étages, le temps d'exécution pour la séquence

(T) peut être déterminé à partir du nombre d'opérateurs en parallèle (P). L'équation suivante sert à calculer le temps d'exécution en termes du nombre de cycles :

$$T = n - 1 + \left\lceil \frac{N}{P} \right\rceil \tag{3-21}$$

où l'opérateur $\lceil \ \rceil$ est défini comme l'arrondissement supérieur ($\lceil 4.1 \rceil = 5$).

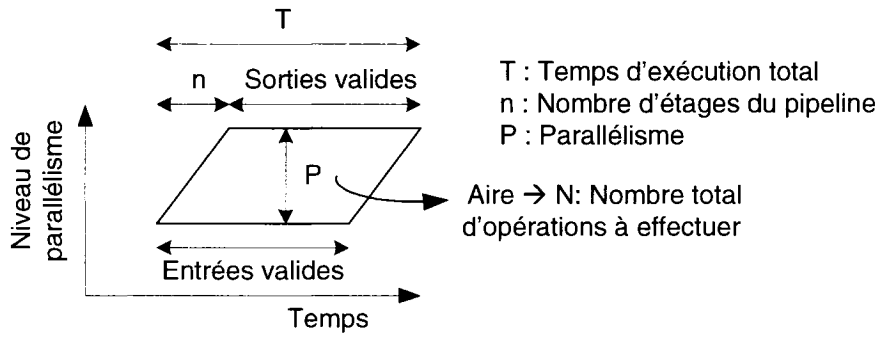

Figure 3.9 : Parallélisation dans un pipeline avec des entrées consécutives

Le nombre d'étages du pipeline est une contrainte imposée par le chemin critique de la chaîne de traitement. Pour conserver la même fréquence d'horloge du processeur de base, il faut par conséquent diviser le chemin critique par la période d'horloge afin de déterminer le nombre d'étages minimal nécessaire au pipeline (équation 3-1). Par contre, il est possible de constater, à l'aide de l'équation pour calculer le temps d'exécution, qu'il faut réduire autant que possible le nombre d'étages du pipeline pour obtenir le temps d'exécution minimal.

Pour étudier l'évolution du gain local, le nombre d'opérations à effectuer (N) est fixé à 64 et le temps d'exécution de référence est fixé à une seule opération (P=1). Le gain local est défini comme le rapport entre le temps d'exécution de référence et le temps d'exécution après optimisation (P augmenté). L'équation 3-22 définit ce concept. À présent, il est possible de déterminer l'évolution du gain local (s) en fonction du niveau de parallélisation et du nombre d'étages du pipeline.

$$T_1 = n-1+N, P=1, N=64$$

$$s = \frac{T_1}{T} = \frac{n-1+N}{n-1+\left\lceil \frac{N}{P} \right\rceil} = \frac{1}{\frac{n-1}{n+N-1} + \frac{1}{(n+N-1)\left\lceil \frac{N}{P} \right\rceil}} \quad \textbf{(3-22)}$$

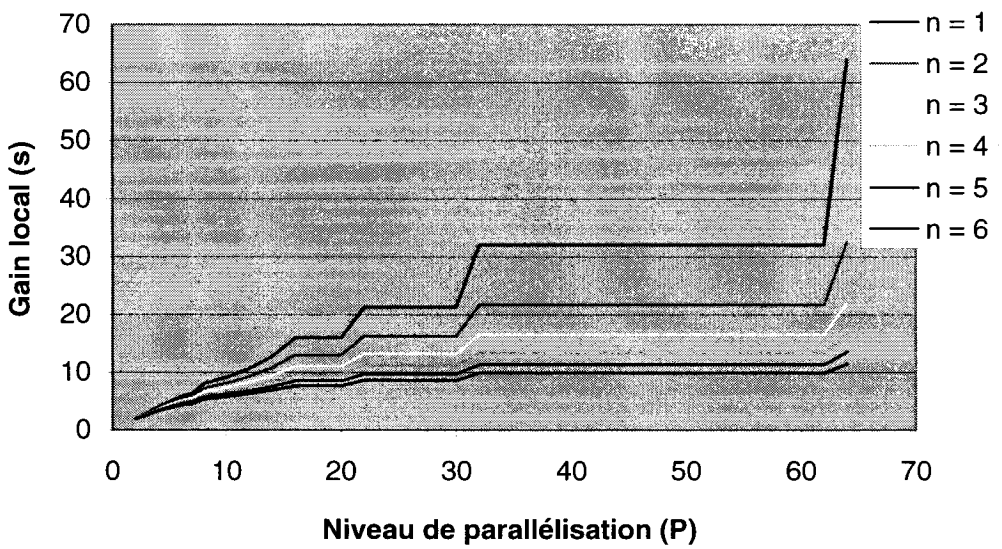

Figure 3.10 : Gain local en fonction du parallélisme pour des entrées consécutives

Ce dernier graphique révèle que le gain local évolue en pallier. En d'autres mots, le gain local est maximal lorsque le niveau de parallélisation est un facteur du nombre d'opérations à effectuer (2,4,8,16,32,64 pour N =64)). Ce phénomène peut s'expliquer par le fait que lorsque plusieurs opérations sont disposées en parallèles et que P n'est pas un facteur de N, plusieurs opérations n'effectuent absolument aucun travail lors du dernier traitement (N modulo P). Ce phénomène de pallier est d'autant plus marquant lorsque le niveau de pipelinage est faible. De plus, dans le cas où n = 1, il est possible d'observer qu'au fur et à mesure que le niveau de parallélisation augmente, le gain local évolue linéairement. Dans les autres cas, le gain local est augmenté par un facteur plus petit que 2 chaque fois où le niveau de parallélisation est doublé. L'augmentation du niveau de parallélisation se traduit par une augmentation de la complexité matérielle.

Dans un second cas, l'analyse peut être portée à des opérations dont les entrées ne sont pas consécutives. Par entrées non-consécutives, il est sous-entendu que le résultat précédent est attendu avant d'injecter de nouvelles données dans l'instruction spécialisée. La figure 3.11 illustre ce concept et l'équation suivante donne le temps d'exécution.

$$T = n\left\lceil \frac{N}{P} \right\rceil \tag{3-23}$$

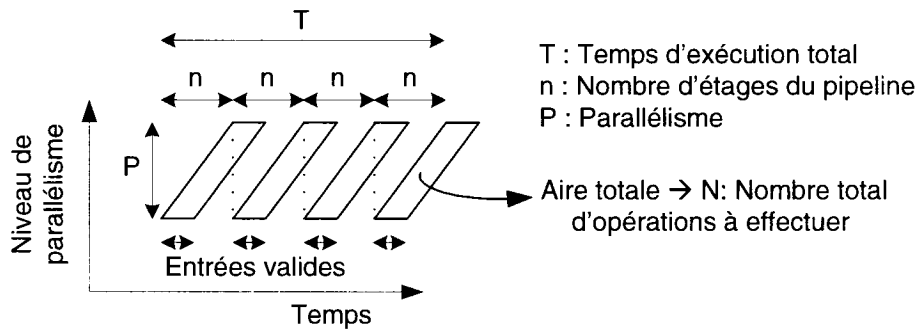

Figure 3.11 : Parallélisation dans un pipeline avec des entrées non-consécutives

Comme dans le cas précédent, le nombre d'opérations à effectuer (N) est fixé à 64 et le temps d'exécution de référence est fixé à une seule opération (P=1). L'équation suivante définit le gain local. L'équation mentionne que le gain local, contrairement au cas précédent, n'est pas influencé par le nombre d'étages du pipeline. Avec ces considérations, il est possible de déterminer l'évolution du gain local (s) en fonction du niveau de parallélisation.

$$T_1 = nN, P = 1, N = 64$$

$$s = \frac{T_1}{T} = \frac{nN}{n\left\lceil \frac{N}{P} \right\rceil} = \frac{N}{\left\lceil \frac{N}{P} \right\rceil} \tag{3-24}$$

Ce dernier graphique révèle que le gain local évolue comme dans le cas précédent, i.e. en paliers. Les mêmes explications s'appliquent dans ce cas en ce qui concerne les paliers. Le gain local de ce type de pipeline évolue linéairement avec le niveau de parallélisation. Il est à noter cependant que le temps d'exécution sera toujours plus petit avec des entrées consécutives.

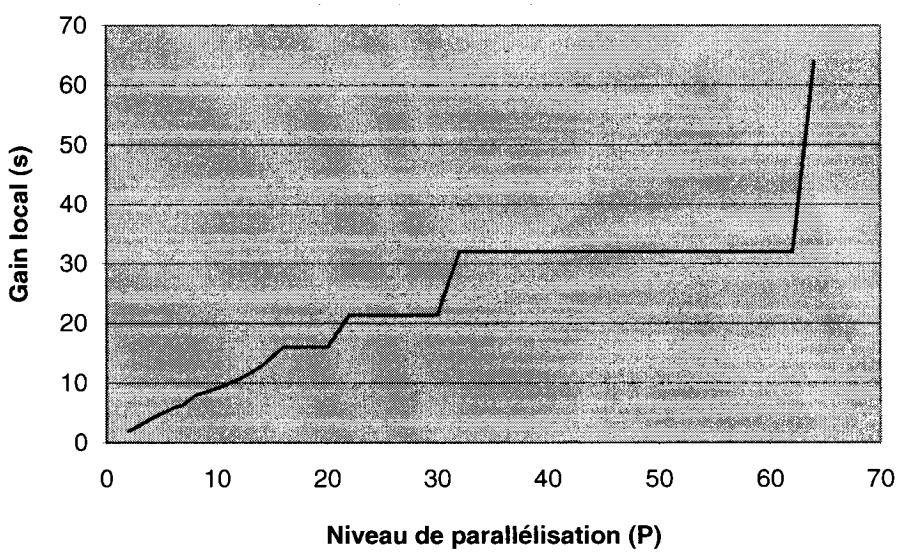

Figure 3.12 : Gain local en fonction du parallélisme pour des entrées non-consécutives

3.3.5 Autres considérations

Dans l'article [40], l'auteur a effectué une recherche concernant les caractéristiques que doivent avoir les instructions spécialisées et les limites de l'accélération. Pour effectuer sa recherche, il a utilisé un banc d'essai commercial, nommé MiBench. L'ensemble des programmes de ce dernier avait des applications dans différents domaines, comme les télécommunications, la sécurité, les réseaux et l'automatisation. Sa stratégie était en premier lieu de diviser tous ces différents programmes en un ensemble de blocs. Ensuite, il a repéré les blocs qui consommaient plus de 95% du temps d'exécution. Sa recherche a été effectuée pour un processeur RISC, avec l'exécution d'une instruction par cycle (« *single-issue* »), pipeliné et avec la supposition que toutes les données se retrouvent

dans l'antémémoire (100% « *cache hit* »). De plus, tous ces programmes ont été compilés avec gcc et avec un niveau d'optimisation 3 (-O3). Aucune opération mémoire n'a été intégrée dans les instructions spécialisées. Pour la génération d'instructions spécialisées, Yu et Mitra [40] ont utilisé un outil qui génère automatiquement des instructions. Toute la recherche repose sur des simulations précises au niveau cycle.

La première exploration a été au niveau du nombre d'entrées et de sorties. Ils ont généré des instructions spécialisées MISO avec deux entrées et une sortie (« *Multiple-Inputs Single-Output* ») et d'autres MIMO (« *Multiple-Inputs Multiple-Outputs* ») sans contrainte d'entrée et de sortie. La conclusion de cette première exploration a été que les instructions spécialisées MIMO donnaient un facteur d'accélération plus élevé et que deux sorties semblaient donner les meilleurs résultats. Par conséquent, le nombre de sorties a été fixé à 2 et le nombre d'entrées a été varié et ce, pour tous les programmes. Suite à cette deuxième exploration, il s'est révélé qu'avec 4 entrées pour les instructions spécialisées, le gain maximal était atteint. Comme troisième exploration, ils ont imposé une contrainte sur l'aire pour les instructions spécialisées (IS). Celle-ci a démontré que dépassé un certain niveau de ressources, soit l'aire de 25 additionneurs, le gain augmentait peu ou pas. Finalement, le nombre d'IS a été contraint à une limite. Cette dernière exploration a laissé paraître, que dans l'ensemble, 5 IS suffisaient pour atteindre le facteur d'accélération maximal.

CHAPITRE 4
MÉTHODOLOGIE

Dans le chapitre précédent, les techniques d'optimisation pour les processeurs configurables ont été présentées et analysées. À présent, il est important d'investiguer quelle est la méthodologie à employer pour obtenir un prototype. Observons d'abord que la conception d'un ASIP est un processus itératif, i.e. que plusieurs configurations et unités spécialisées peuvent être générées avant d'en arriver au prototype. La méthodologie présentée dans le présent chapitre est basée sur une technologie de processeurs configurables à noyau fixe et paramétrable. Cette méthodologie a été inspirée de celle proposée par Quinn et al. dans [44].

4.1 Méthodologie de conception de base

La méthodologie de conception de base peut être principalement divisée en quatre phases. La séquence des différentes phases est illustrée à la figure 4.1. En premier lieu, il faut écrire l'application sous forme logicielle avec un langage de programmation. Après le développement logiciel de l'application, il faut ensuite considérer les paramètres du noyau du processeur. Ces paramètres peuvent avoir une influence considérable sur l'exécution de l'application, il faut par conséquent bien saisir leur impact sur les performances. Ces paramètres concernent principalement la taille des antémémoires, des unités fonctionnelles et du nombre de registres généraux. Après que l'architecture de base du processeur soit déterminée, il faut ajouter une unité spécialisée au noyau afin d'accélérer l'exécution de l'application. Cette phase consiste à concevoir des instructions spécialisées à partir du code de l'application. Dans cette phase, nous mettrons à profit les techniques d'optimisations du chapitre précédent. Finalement, la dernière phase consiste à l'implémentation du prototype à partir de la description HDL du processeur.

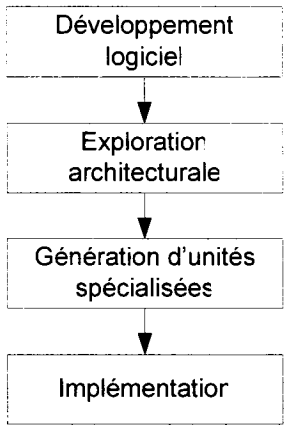

Figure 4.1 : Méthodologie de conception pour les processeurs configurables

Les différentes phases de la méthodologie de conception seront étudiées plus en profondeur dans les prochaines sections.

4.2 Développement logiciel

La première phase de la méthodologie peut être divisée en deux sous-sections, soit le développement de l'application sous forme logicielle et l'optimisation. La première consiste à avoir un modèle fiable de la fonctionnalité en langage de programmation. Pour ce faire, la plupart des technologies de processeur configurable utilisent le langage C et C++. Cette première étape est habituellement relativement courte. Elle ne vise qu'à effectuer une première ébauche du programme sans penser aux optimisations possibles.

4.2.1 Optimisation logicielle

La phase d'optimisation logicielle est en premier lieu un recodage du programme afin de tirer profit des différentes unités fonctionnelles. Il est à mentionner que l'optimisation logicielle avec le langage C est plus facile qu'avec du C++. En fait, le langage C est un

langage de moins haut niveau et se rapproche beaucoup plus de l'assembleur, ce qui est plus facile à optimiser. Il peut être possible d'atteindre jusqu'à un facteur 2 en accélération juste par un recodage de l'algorithme.

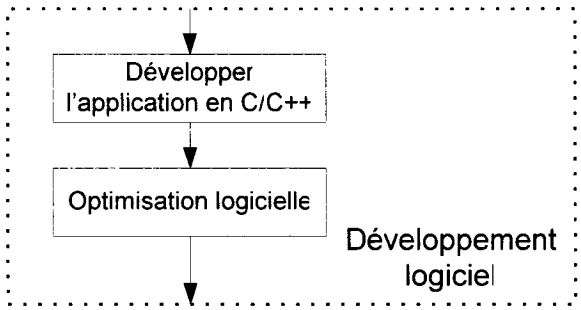

Figure 4.2 : Phase de développement logiciel

L'une des techniques d'optimisation souvent utilisée est le déroulement de boucle, cette technique permet aussi la parallélisation. Cette façon de faire permet de réduire le nombre d'itérations d'une boucle et de ce fait, le nombre d'instructions requises pour le bouclage. Une autre technique consiste au déplacement de code invariant hors des boucles. En d'autres mots, le code qui ne change pas d'une itération à une autre d'une boucle doit être déplacé en dehors de la boucle afin de ne pas être exécuté de multiples fois. À ces deux dernières techniques, il faut joindre la simplification des expressions mathématiques et logiques. Par exemple, il est possible de remplacer des multiplications et des divisions par des puissances de 2 par de simples décalages de bits. De plus, il est possible de faire de l'alignement de fonction. Cette technique a pour conséquence de copier le code d'une fonction à l'endroit invoqué dans le programme, ceci évite les cycles de branchement pour un appel de fonction. Un autre facteur important est l'accès des données dans un tableau. S'il faut parcourir un tableau multidimensionnel, il faut accéder les données dans l'ordre où elles ont été emmagasinées en mémoire afin de réduire les échecs d'accès aux antémémoires (« *cache miss* »). Avec un langage de programmation comme le C/C++, il

faut accéder les indices de droite en premier. Pour de plus amples informations sur les optimisations logicielles, le lecteur est invité à consulter l'annexe D.

Il est à noter que les paramètres du compilateur peuvent affecter considérablement les performances. Ainsi, il faut spécifier au compilateur que le programme devra être compilé avec un niveau d'optimisation 3 (-O3, les optimisations les plus agressives que le compilateur peut effectuer [2]). Par la suite, il faut toujours vérifier la fonctionnalité du programme. Si le compilateur prend en paramètre l'architecture du processeur, il faut aussi lui spécifier. Il se peut qu'il y ait plus d'un compilateur pour un même processeur, comme c'est le cas pour le Xtensa [52] (xt-xcc et xt-gcc). Ainsi, il est recommandé de vérifier l'exécution avec les deux compilations afin de vérifier lequel des compilateurs est le plus performant.

4.3 Exploration architecturale de processeur

Comme mentionné précédemment, cette méthodologie est consacrée aux processeurs configurables à noyau fixe et paramétrable. Ainsi, après avoir développé l'application sous forme logicielle, il faut explorer quels sont les paramètres adéquats à notre application. Cette phase est en fait itérative, il s'agit de sélectionner les paramètres selon l'expérience du concepteur et ensuite d'effectuer un profilage afin de déterminer les performances. Après une première itération, s'il persiste un doute quant à la validité des paramètres du processeur, il faut essayer avec différents paramètres afin de déterminer les paramètres optimaux. Avec la technologie Xtensa, cette recherche procède par essais et erreurs. Parmi les paramètres que l'on peut ajuster, il y a le nombre de registres, la taille et le type d'antémémoire, et la présence de diverses unités opératives comme une unité de multiplication, une unité de division et une unité virgule flottante. La figure 4.3 illustre le flot de conception de la phase d'exploration architecturale d'un processeur.

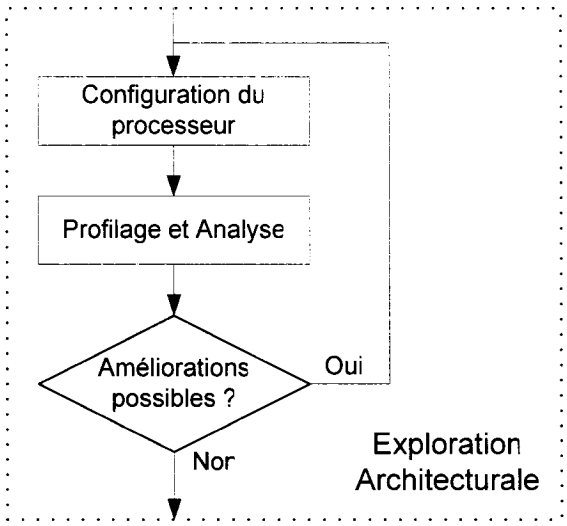

Figure 4.3 : Exploration architecturale du processeur de base

4.3.1 Configuration du processeur

Il faut tenir compte de plusieurs facteurs pour obtenir le nombre de registres physiques optimal du processeur. En premier lieu, il faut déterminer quelle est la taille du fenêtrage des registres (Nios → 32, Xtensa → 16) du processeur. De plus, il faut tenir compte du mécanisme de déplacement de la fenêtre. À chaque appel de fonction ou d'interruption, la fenêtre des registres est déplacée d'une certaine distance, qui peut être fixe ou variable. Dans le cas du processeur Nios, le déplacement est constant et d'une distance de 16 registres. Pour ce qui est du processeur Xtensa, le déplacement de la fenêtre est variable, d'une distance de 4, 8 ou 12 et déterminé par l'appelant. De plus, le nombre de registres dépend du niveau d'imbrication des fonctions et du nombre d'interruptions de l'application. L'équation suivante permet de calculer le nombre de registres physiques nécessaires pour éviter un débordement des registres (« *overflow* ») lorsque la distance de déplacement est constante :

$$R = T + (N + I)D \tag{4-1}$$

où

R : Nombre de registres physiques nécessaires
T : Taille de la fenêtre
I : Nombre d'interruptions matérielles possibles
N : Niveau d'imbrication maximale des fonctions
D : Distance du déplacement de la fenêtre

Par niveau d'imbrication des fonctions, il est sous-entendu que le programme principal est le niveau 0 et que toute fonction appelée constitue un nouveau niveau d'imbrication. Ainsi, si une fonction appelée par le programme principal appelle elle-même une autre fonction, cela implique un niveau d'imbrication de 2. Par exemple, si un processeur utilise une fenêtre de 32 registres avec une distance de déplacement de 16, que le processeur n'a aucune interruption et que le niveau d'imbrication maximal des fonctions est de 2, il faut par conséquent 64 registres physiques au processeur pour éviter les débordements (32 + (2+0)16 = 64). Si la distance de déplacement des registres est variable, il s'agit de prendre la distance de déplacement la plus longue dans le calcul pour obtenir un bon estimé. De plus, il est possible à l'aide d'un profilage de vérifier s'il y a eu un débordement des registres. Un débordement des registres nécessite une exception (sauvegarde des registres sur la pile), ce qui est enregistré dans un profilage.

Tel que mentionné plus tôt en ce qui concerne les unités fonctionnelles, il est possible d'ajouter une unité de multiplication, une unité de division, un multiplieur accumulateur (MAC) et une unité arithmétique virgule flottante. Ces unités valent la peine d'être ajoutées à l'architecture si ces fonctionnalités sont utilisées de façon fréquente dans l'application. Bien entendu, une unité virgule flottante doit être ajoutée si les données à manipuler sont dans ce format. Une autre option qui est possible de retrouver est un bouclage sans instructions de surcharge (« *Option Enable Zero-Overhead Loop Instructions* »). Cette option permet d'effectuer des boucles en réduisant le nombre d'instructions relatives aux compteurs et aux branchements. Celle-ci est très pratique lorsque l'application contient un grand nombre de boucles ou de longues boucles (plusieurs itérations).

En ce qui concerne l'antémémoire, sa taille et son type peuvent avoir un fort impact sur les performances. Ainsi, il est important de configurer l'antémémoire pour les instructions et les données adéquatement. Généralement, une application bénéficie de l'augmentation de la taille de l'antémémoire jusqu'au point où les performances saturent. Quand ce point est atteint, il se produit l'un des deux phénomènes suivants : soit que l'antémémoire est de taille suffisante pour que l'application n'ait jamais à chercher de l'information de la mémoire principale (RAM ou ROM) ou que l'application accède les données de façon aléatoire et que l'augmentation de l'antémémoire n'augmente plus de manière significative la chance de trouver localement la prochaine information demandée. Il est relativement rare de trouver une application qui n'offre aucun gain suite à l'inclusion d'une antémémoire car la plupart des applications démontrent une certaine localité des données. Une technique simple pour connaître la taille optimale de l'antémémoire est de commencer avec le maximum d'antémémoire que possible et d'en diminuer la taille jusqu'à temps que les performances se dégradent considérablement. De plus, il est proposé de faire varier l'antémémoire d'instructions avant celle des données. Cette façon de faire permet de fixer la taille de l'une avant d'étudier l'impact de la taille de l'autre.

En plus de la taille, certaines technologies permettent de sélectionner le type d'antémémoire, i.e. à correspondance directe, associative par ensemble de deux blocs, trois blocs ou quatre blocs. De plus, il est possible de spécifier la politique d'écriture, soit l'écriture simultanée (« *write through* ») ou la réécriture (« *write back* »). Chacune de ces techniques offre des avantages spécifiques. De plus, il est possible de sélectionner la taille des blocs (« *line size* », 16, 32 ou 64 octets). L'augmentation la taille des blocs a pour conséquence de réduire le taux d'échec, mais elle augmente la pénalité d'échec (chargement plus long). Il existe une règle du pouce qui dit qu'une antémémoire à correspondance directe de taille N a environ le même taux d'échec qu'une antémémoire associative à deux blocs par ensemble de taille N/2 [22]. Il est à considérer que si la taille

de l'antémémoire est assez élevée pour contenir tout le code de l'application, le type de mémoire aura peu ou pas d'impact.

4.3.2 Profilage

Pour profiler l'application, il faut a priori un programme codé en C/C++ ou en assembleur et un flot de données typique pour l'application. Le programme de l'application doit être avant tout compilé pour le processeur de base de l'application. Après la compilation, le programme doit être exécuté afin de déterminer combien de cycles chaque section du code prend. Pour ce faire, il y a deux approches pouvant être adoptées : 1) l'une basée sur la simulation et 2) une autre basée sur une plate-forme matérielle.

L'approche basée sur la simulation est réalisée à l'aide d'un simulateur de jeu d'instruction (ISS) et d'un modèle matériel du système. Le modèle matériel du système comprend l'architecture mémoire, un modèle du pipeline, les différents types d'unités opératives, le nombre de registres, la taille et le type d'antémémoire. Le simulateur utilise le modèle du processeur afin d'anticiper le comportement du système réel. Cette façon de faire permet d'obtenir rapidement des résultats de profilage assez précis, car la simulation est effectuée à un niveau assez élevé. De plus, le profilage des fonctions à l'interne d'une autre n'influence pas le résultat de la fonction externe. En d'autres mots, la délimitation des sections du code n'influence pas les résultats de profilage. De plus, il est possible d'obtenir un arbre des fonctions et des informations comme le nombre de fois qu'une fonction est appelée, ainsi que combien de fois elle est appelée par chaque appelant. Cette technique est utilisée avec la technologie Xtensa.

Une seconde approche consiste à faire un profilage dynamique, c'est-à-dire que le programme est exécuté sur une plate-forme avec un processeur de base. Cette approche est souvent réalisée à l'aide d'un FPGA, sur lequel il est possible de modifier l'architecture du processeur. L'information fournie par le profilage est moins précise que

celle basée sur l'approche par simulation. En fait, le profilage sur une plate-forme matérielle nécessite d'exécuter des instructions afin de lire les valeurs d'un compteur de temps et de calculer le nombre de cycles exécutés. Ainsi, ces instructions exécutées pour le profilage introduisent des erreurs. Ces erreurs demeurent minimes si le nombre de cycles d'exécution est élevé pour la fonction à profiler par rapport aux instructions utilisées pour profiler. C'est ce genre d'approche qui est adoptée avec le processeur Nios, car aucun modèle logiciel du processeur n'est fourni avec la technologie. Ceci constitue une lacune de la technologie Nios. Par contre, il est possible de simuler le processeur Nios au niveau VHDL avec *Modelsim*.

La figure 4.4 résume les deux approches.

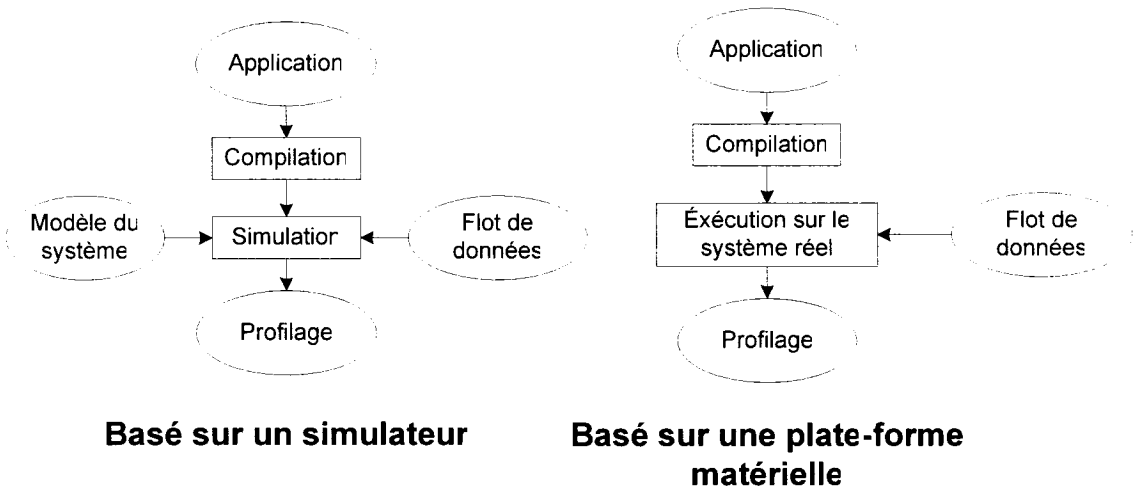

Figure 4.4 : Processus du profilage, basé sur un simulateur ou sur une plate-forme matérielle

4.4 Génération d'unités spécialisées

Après avoir sélectionné l'architecture optimale du processeur, il faut à présent concevoir l'unité spécialisée pour l'application. Avec l'information du profilage, il faut par la suite identifier les sections chaudes du programme, c'est-à-dire les sections qui consomment la plus importante portion du temps d'exécution. Une fois que ces sections ont été

identifiées, il faut avec la fonctionnalité de ces sections, en extraire des patrons de fonctionnalité. Ces patrons de fonctionnalité seront par la suite utilisés pour générer des instructions spécialisées. Les instructions spécialisées sont généralement soit des sous-routines de l'application souvent utilisées ou une séquence d'instructions qui est commune dans l'exécution de l'application [32]. La figure 4.5 illustre le flot de conception d'unités spécialisées.

La génération d'instructions spécialisées peut être effectuée manuellement ou de façon automatique. La conception manuelle repose sur l'expérience du concepteur, c'est-à-dire que le concepteur génère des instructions spécialisées qui lui semblent être profitables pour l'application. Pour sa part, la conception automatisée d'instructions spécialisées repose sur un outil qui extrait les patrons d'instruction qui peuvent être profitables pour une application.

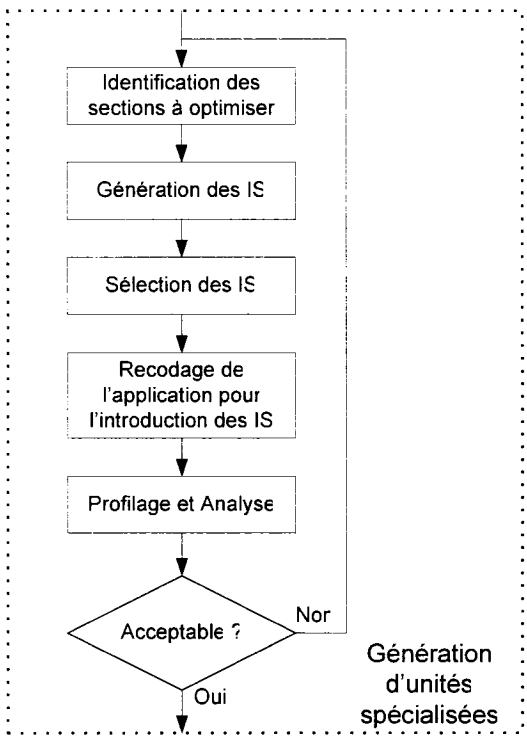

Figure 4.5 : Génération d'une unité spécialisée

Suite à la génération d'instructions, il faut sélectionner celles qui permettront plus facilement d'atteindre les objectifs. Cette sélection dépend des contraintes en termes du nombre d'instructions, de l'aire de l'unité spécialisée et des objectifs visés en performance. L'introduction d'instructions spécialisées (IS) impose un recodage de l'application afin de spécifier les endroits où elles seront bénéfiques. Ensuite, un profilage doit être effectué afin de déterminer de nouveau les performances avec les IS sélectionnées.

4.4.1 Conception manuelle

L'efficacité du processus manuel repose sur l'expérience du concepteur. Bien entendu, les sections qui consomment le plus de cycles doivent avant tout être ciblées pour la réalisation d'instructions spécialisées. Une fois que la section à optimiser est bien identifiée, il faut la décortiquer afin de reconnaître les dépendances entre les instructions. En fait, le concepteur doit bien maîtriser la fonctionnalité de la section à optimiser avant de commencer à élaborer des instructions spécialisées.

Les sections de code qui s'apprêtent le mieux à l'optimisation par IS sont celles qui manipulent des données. Ces sections sont en fait limitées par des instructions de branchement. Cependant, il est possible de déplacer ces limites comme par un déroulement de boucles ou alignement de fonctions. Le concepteur peut modifier l'algorithme à sa guise en autant qu'il en conserve la cohérence, bref que la modification n'influence pas le résultat. Ces modifications impliquent qu'il soit possible de regrouper toutes les instructions de chargement de données au début d'une fonction et tous les enregistrements à la fin de celle-ci. Les techniques d'optimisation du chapitre 3 peuvent être utilisées afin d'exploiter au maximum le parallélisme de l'application. De plus, il ne faut pas perdre de vue qu'il est toujours préférable d'accélérer le cas le plus fréquent.

4.4.2 Conception automatique

Cette section vise à expliquer brièvement le processus de génération automatique d'IS et la majorité de l'information est tirée des articles [5] [11][39]. Ce processus se divise principalement en deux étapes : soit l'énumération des patrons et la sélection optimale d'un ensemble de patrons. La première étape à effectuer dans l'énumération des patrons est l'établissement d'un DFG (« *Dataflow Graph* ») des sections à optimiser. Un DFG représente les relations de dépendance de données entre les différentes opérations dans une section de code. La figure 4.6 illustre un exemple de DFG. Un DFG est composé d'un ensemble de nœuds, chacun étant associé à une opération, et d'un ensemble de relations de dépendances (flèches). Dû à certaines contraintes, ce ne sont pas tous les nœuds qui peuvent être inclus dans un patron. Par exemple, les opérations d'accès mémoire et celles de transfert de contrôle (branchement) ne sont typiquement pas incluses dans les patrons. Par conséquent, un nœud est dit valide s'il peut être inclus dans une instruction spécialisée, sinon, il est invalide.

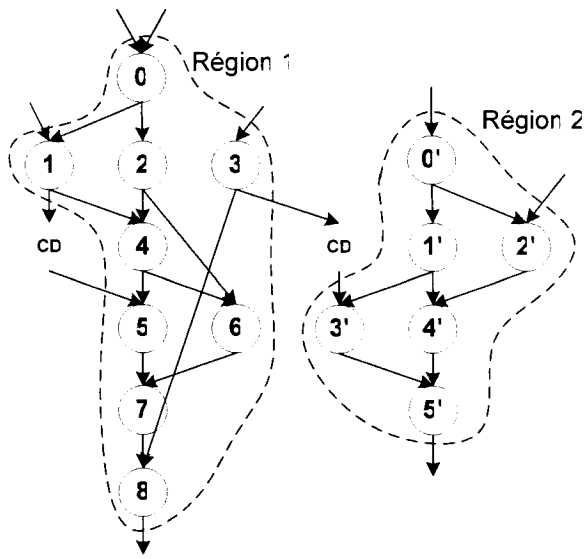

Figure 4.6 : Exemple de DFG pour l'énumération de patrons de fonctionnalité

Le DFG peut être divisé en plusieurs régions. Une région est définie comme un ensemble de noeuds 1) qui contient juste des nœuds valides, 2) dont il existe un chemin indirect entre chaque paire de nœuds et 3) dont il n'existe pas de relation de dépendance entre un des nœuds de la région et un nœud valide externe. Un chemin est dit indirect lorsqu'il est constitué d'un ensemble de relations qui peuvent autant être dans le sens des flèches que dans celui inverse. Les nœuds invalides n'appartiennent à aucune région. Une fois que les régions ont été délimitées, un algorithme d'identification des instructions spécialisées est appliqué sur chacune des régions. Un patron de fonctionnalité appartient par conséquent à une région.

Lors de la génération, les patrons évalués doivent avant tout satisfaire certaines contraintes. Les contraintes courantes sont le nombre d'entrées, le nombre de sorties. De plus, le patron doit être convexe (défini plus loin) et connecté. Pour ce qui est des entrées, il suffit d'identifier le nombre de relations dont dépend le patron. Par exemple, si la contrainte est fixée à 3 entrées, le patron des nœuds 7-8 de la figure 4.6 respecte cette contrainte, mais le patron 5-7-8 ne la respecte pas (4 entrées). En ce qui attrait aux sorties, il suffit d'identifier le nombre de relations sortantes du patron. Par exemple, si la contrainte est fixée à 2 sorties, le patron des nœuds 0-1-2 de la figure 4.6 respecte cette contrainte, mais le patron 0-1-2-4 ne la respecte pas (3 sorties). Pour qu'un patron soit convexe, il faut qu'il n'y ait aucun chemin direct entre deux nœuds du patron qui passe par un nœud externe au patron. Cette dernière contrainte se traduit par le fait qu'il faut qu'aucune entrée du patron ne dépende d'une de ses sorties. Par exemple, le patron des nœuds 4-5-7 de la figure 4.6 n'est pas convexe, car l'une de ses sorties doit passer par le nœud 6 et qu'une de ses entrées provient de ce nœud. Par conséquent, ce patron n'est pas réalisable. La contrainte de connectivité stipule qu'il doit y avoir un chemin indirect entre toutes les entrées et sorties à l'intérieur du patron. Par exemple, le patron 0-1-2 est connecté, mais le patron 0-1-2-3 ne l'est pas, car le nœud 3 est complètement disjoint des autres. La seule vraie contrainte parmi celles-ci est la convexité d'un patron, les autres ne sont établis que pour réduire l'étendue des possibilités (espace d'exploration)

Après avoir généré un ensemble de patrons, il faut sélectionner quels seront ceux qui seront introduits dans le processeur. L'algorithme de sélection des IS évalue le potentiel de chacun des patrons. Cet algorithme vise à atteindre un objectif sous une certaine contrainte. La plupart du temps, l'objectif visé est une augmentation des performances sous une contrainte d'aire. En premier lieu, chacun des candidats doit être évalué à l'aide d'une métrique. Cette métrique déterminera quels sont les candidats qui ont le plus de potentiel à aider pour atteindre l'objectif. Ensuite, les candidats seront évalués un à un, commençant par le meilleur. Un autre facteur qui doit être pris en considération est le chevauchement. En d'autres mots, la fonctionnalité d'une IS ne doit pas chevaucher celle d'une autre. Bref, il n'est pas possible d'utiliser deux IS pour la même fonctionnalité en même temps. Les candidats seront retenus ou éliminés tout dépendant s'ils respectent les contraintes.

Il est à remarquer que la plupart des articles publiant des résultats avec des IS générées automatiquement démontrent de faible accélération, moins que 2 (citons en exemple l'article [40]). La faiblesse de l'accélération provient du fait que l'algorithme se contente d'identifier les régions bornées par des accès mémoire et des transferts de contrôle sans en modifier les limites. En fait, les accès mémoire peuvent être déplacés au début d'une section et il est possible de modifier légèrement les transferts de contrôle. En d'autres mots, les régions peuvent être agrandies manuellement si la cohérence de l'algorithme est conservée.

4.5 Implémentation

La dernière phase de la méthodologie est celle de l'implémentation. Cette phase consiste à passer de la description matérielle à la réalisation matérielle. La figure 4.7 illustre le flot de conception de cette dernière phase. Avant la synthèse, il faut générer la description HDL du processeur, cette description peut être en VHDL ou Verilog (tout dépendant de la technologie). Après la génération de la description HDL, il faut transformer la

description en un circuit réel pour une technologie ASIC ou FPGA. Pour les technologies FPGA, les outils de synthèse et de placement/routage sont souvent intégrés comme un tout.

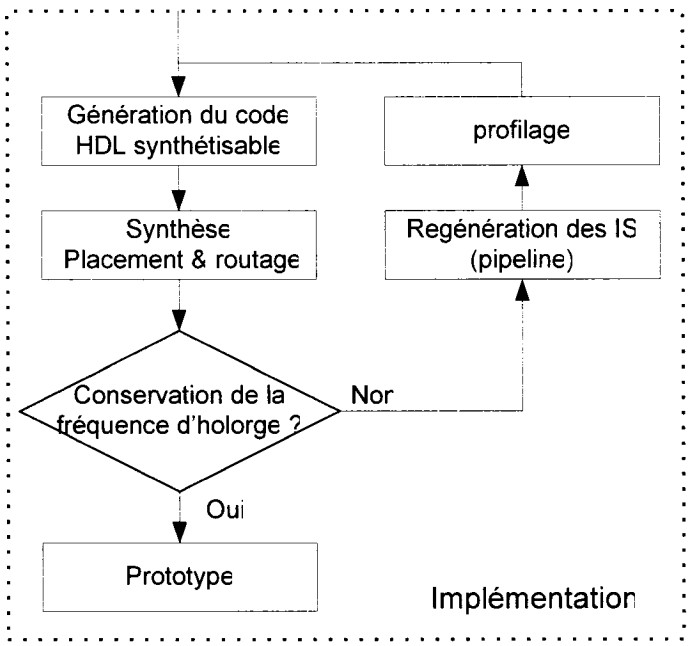

Figure 4.7 : Phase d'implémentation

Pour ce qui est de la technologie ASIC, il est possible d'effectuer une synthèse à l'aide d'outils standard comme le *Synopsys Design Compiler*. Certaines technologies de processeur configurable comme celle du Xtensa fournissent des scripts flexibles pour faire cette étape. Ces scripts sont paramétrables et permettent de sélectionner les contraintes, l'effort de compilation, les rapports, le nombre de passe d'optimisation, les bibliothèques et les cellules. De plus, il est possible d'utiliser des outils comme le *Synopsys Physical Compiler* qui permet d'obtenir de meilleurs résultats quant au chemin critique. Cet outil effectue la synthèse et le placement des cellules de façon concurrente pour mieux atteindre la contrainte en termes de fréquence. Avec cet outil, il est possible d'atteindre jusqu'à 25% d'amélioration des performances pour une pénalité en aire

négligeable. Le routage est ensuite réalisé avec des outils comme *Cadence Silicon Ensemble*.

À la fin de la synthèse et du placement/routage, il est possible de déterminer l'emplacement du chemin critique du processeur. S'il se retrouve dans l'unité spécialisée, la fréquence d'horloge à laquelle le processeur pourra opérer sera inférieure à celle de la configuration de base. En fixant la fréquence d'horloge du processeur à celle de la configuration de base, cela évite de ralentir des sections de l'application pour en accélérer d'autres. Cette méthodologie propose donc de conserver la fréquence d'horloge de la configuration de base du processeur. Par contre, il est à noter qu'il est possible d'obtenir un gain en performance même si la fréquence du processeur n'est pas la même que celle du processeur de base.

Si la contrainte de fréquence d'horloge n'est pas rencontrée, il faut régénérer l'unité spécialisée. En fait, cette séquence consiste à pipeliner l'instruction spécialisée dans laquelle se retrouve le chemin critique. Ainsi, le chemin critique sera scindé en plusieurs morceaux. Ensuite, il faudra effectuer un nouveau profilage afin de déterminer le nombre de cycles d'exécution avec la nouvelle unité spécialisée. De plus, il faudra effectuer de nouveau la synthèse et le placement/routage pour déterminer où se retrouve le chemin critique de l'application après la régénération de l'unité spécialisée. Ce processus doit continuer jusqu'à ce que le chemin critique ne se retrouve plus dans l'unité spécialisée.

CHAPITRE 5

RÉSULTATS

Le présent chapitre est consacré à l'explication de l'implémentation des égaliseurs avec le processeur Xtensa et le Nios. Le détail des instructions spécialisées développées dans le cadre de l'implémentation avec le processeur Xtensa est d'abord présenté. Par la suite, une analyse est réalisée en ce qui a trait à la complexité matérielle, la fréquence des égaliseurs et le rapport des produits AT des processeurs spécialisés. Finalement, les résultats obtenus avec l'implémentation pour le processeur Nios sont présentés. Ces résultats sont axés comme pour le processeur Xtensa sur la complexité matérielle, les performances et sur le rapport AT.

5.1 Implémentation avec le processeur Xtensa

Pour obtenir un gain en performance, l'une des premières techniques d'optimisation a été de repérer les variables redondantes sur lesquelles des traitements sont effectués et de leur dédier des registres spécialisés. Dans le cadre de notre application, ceci s'est traduit par des registres spécialisés pour tous les coefficients, les données du filtre, ainsi que pour l'erreur de l'égaliseur. Cette optimisation permet d'éliminer l'accès à la mémoire pour ces variables. En second lieu, plusieurs opérateurs ont été disposés en parallèle pour accélérer les sections d'algorithme exigeantes en nombre de cycles d'exécution, dont celle du filtrage et de la mise à jour des coefficients. Chaque opérateur effectuant un traitement sur différents registres spécialisés. De plus, les opérateurs complexes comme la multiplication ont été partagés entre les différentes instructions spécialisées. Ce partage d'opérateur permet une réduction de la complexité matérielle. Dans la conception des instructions spécialisées, l'objectif, en ce qui concerne la fréquence d'opération, était celle du processeur de base. Ainsi, il n'était pas désiré de réduire la fréquence d'horloge

du processeur suite à l'ajout d'instructions spécialisées. Pour ce faire, certaines instructions spécialisées ont été ordonnancées sur deux cycles d'exécution et pipelinées.

Afin d'optimiser le processeur Xtensa pour l'égalisation LTE-LMS, 5 instructions ont été ajoutées au processeur. A priori, l'algorithme a été divisé en quatre différentes sections, soit le filtrage, la prise de décision et le calcul de l'erreur, le calcul du facteur de correction (μe) et la mise à jour des coefficients. Pour plus de renseignements sur l'algorithme d'égalisation, le lecteur est invité à consulter l'annexe A. Pour la prise de décision et le calcul d'erreur, deux instructions ont été conçues, l'une lorsque l'égaliseur est en entraînement et une autre lorsqu'il est en poursuite. Pour le filtrage, l'instruction spécialisée a été conçue à partir d'un multiplieur-accumulateur (ou « *multiply-accumulate* ») complexe (MAC). Trois variantes de cette instruction ont été implémentées, soit avec un, deux ou quatre multiplieurs complexes (MC) en parallèle. La mise à jour des coefficients est réalisée à l'aide d'une instruction qui est implémentée aussi selon trois variantes avec des nombres différents de multiplieurs complexes.

Dans les instructions précédemment mentionnées, les plus critiques sont celles qui réalisent plusieurs multiplications. Il s'agit du filtrage et de la mise à jour des coefficients. Ce sont principalement ces instructions qui permettent d'obtenir des gains de performance importants. Celles-ci nécessitent plusieurs multiplieurs 16 bits. Par conséquent, il est important de partager les multiplieurs entre les différentes instructions pour que ceux-ci soient utilisés à leur pleine efficacité. Pour la réalisation de l'égaliseur, toutes les données et les coefficients sont contenus dans des registres qui leur sont dédiés. Les entrées des multiplieurs sont donc multiplexées. Il est possible d'observer sur la figure 5.1 que le premier étage des pipelines pour ces deux instructions est identique. La figure 5.2 illustre le second étage du pipeline de l'instruction de filtrage. Cette dernière prend la forme de deux accumulateurs, soit un pour la partie réelle et un autre pour la partie imaginaire. La figure 5.3, pour sa part, représente le dernier étage du pipeline pour l'instruction de mise à jour des coefficients. Ce dernier étage effectue l'addition d'une correction sur les parties réelle et imaginaire du coefficient.

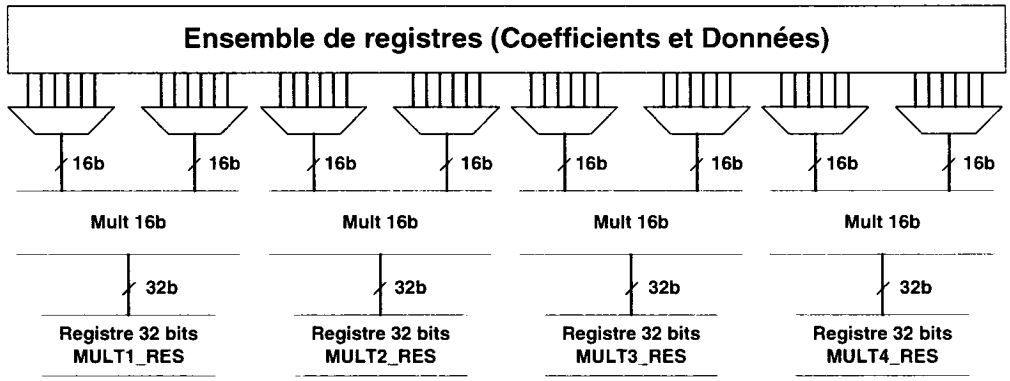

Figure 5.1 : Premier étage du pipeline pour le filtrage et la mise à jour des coefficients

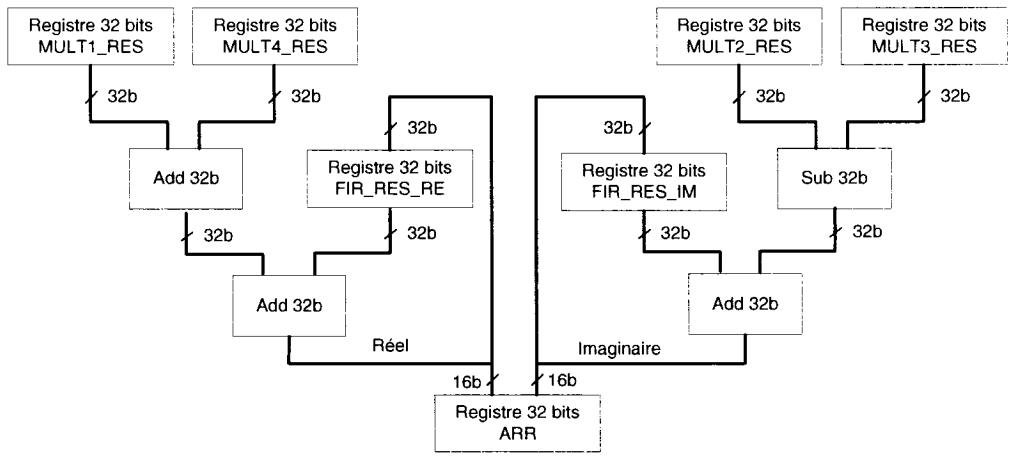

Figure 5.2 : Dernier étage du pipeline pour le filtrage

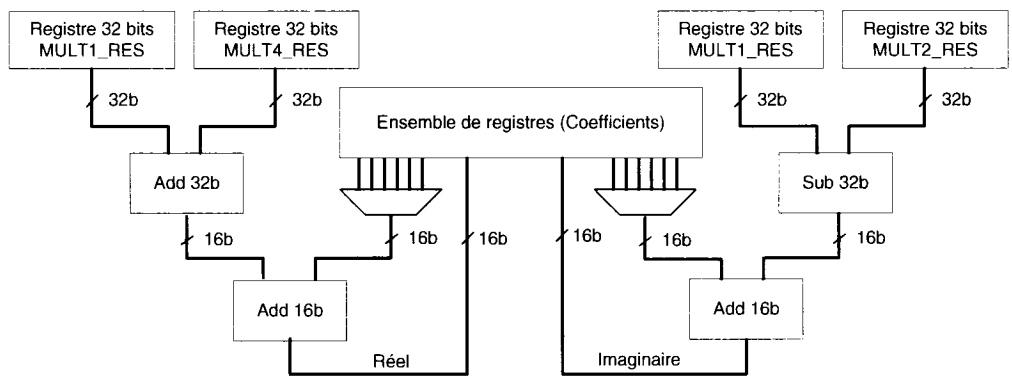

Figure 5.3 : Dernier étage du pipeline pour l'instruction de mise à jour des coefficients

Ces dernières structures ont été implémentées pour le filtrage et la mise à jour des coefficients. À présent, les deux dernières structures vont être étudiées. La première est la prise de décision sur le symbole. Comme mentionné précédemment, l'égaliseur implémenté est pour une modulation BPSK, donc cette instruction ne s'applique que pour ce type de modulation. Celle-ci est illustrée à la figure 5.4. Cette instruction prend en entrée un registre de 32 bits, dont les 16 MSB correspondent à la partie réelle et les 16 LSB à la partie imaginaire (registre ARS). Cette entrée est en fait une estimation du symbole qui se trouve à être la sortie du filtre. Si la partie réelle de la sortie du filtre est positive, alors la décision sera BPSK_P, sinon BPSK_N. Ces valeurs correspondent aux décisions +1 et −1 pour une modulation BPSK. La sortie du premier multiplexeur est alors la décision sur le symbole. Au deuxième multiplexeur, la sortie dépend du présent mode de l'égaliseur, soit en poursuite ou en entraînement. Si l'égaliseur est en mode d'entraînement (TRAIN opcode), la sortie sera une donnée de la séquence d'entraînement, sinon elle sera la décision prise sur le symbole. Le registre de sortie (ARR) aura par conséquent le symbole décidé (réelle 16 MSB, imaginaire 16 LSB). À la suite de ce traitement, l'erreur entre l'estimation et la décision du symbole est calculée. La section de droite calcule la partie réelle de cette erreur et la section de gauche, la partie imaginaire.

De plus, s'il était désiré de modifier le type de modulation, il ne s'agirait que de modifier l'instruction pour la prise de décision et le calcul de l'erreur. Ainsi, les résultats obtenus sont présentés en bit par seconde, car la forme de modulation utilisée est le BPSK. Cependant, les résultats peuvent être présentés sous forme de symboles par seconde. Cette instruction aurait pu être étendue à une modulation QPSK et QAM. Pour réaliser cette instruction pour une modulation QPSK, il aurait suffit de considérer le bit de signe de la partie imaginaire (ARS[15]) et de reproduire la section de droite avec des multiplexeurs pour la section imaginaire (remplacer le 0).

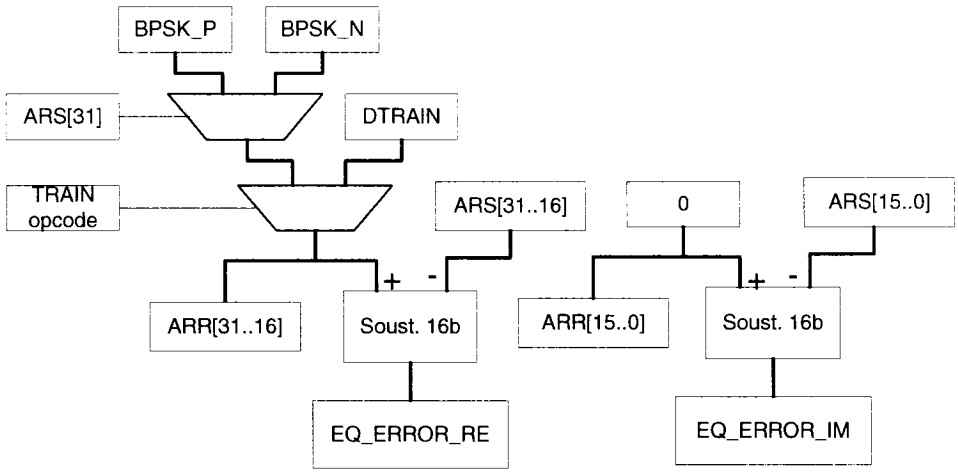

Figure 5.4 : Instruction spécialisée pour la prise de décision et le calcul d'erreur

La dernière instruction spécialisée est utilisée pour calculer le produit de l'erreur par le pas d'adaptation. Celle-ci consiste simplement en deux multiplications, donc elle ne produit pas un très grand facteur d'accélération, par contre elle ne nécessite pas une grande complexité matérielle. Cette instruction est illustrée à la figure 5.5.

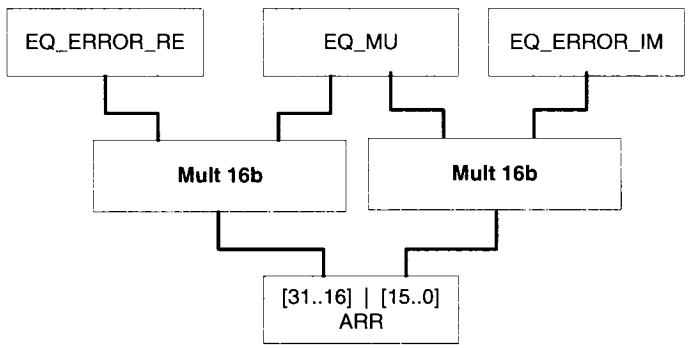

Figure 5.5 : Instruction spécialisée pour le calcul du facteur d'adaptation

Pour la conception du DFE-LMS, 7 instructions ont été ajoutées au processeur Xtensa. Les 5 instructions pour le LTE-LMS s'y retrouvent intégralement. Pour le filtrage dans une DFE, il faut un filtre arrière de plus. Il était donc pertinent de rajouter une instruction

pour ce traitement. À ce filtre arrière, il faut joindre la mise à jour des coefficients, ce qui nécessite une autre instruction.

5.1.1 Complexité matérielle

Pour effectuer la présente étude, six différentes versions du processeur ont été utilisées pour chaque égaliseur. Trois versions ont été réalisées avec un Xtensa sans instruction spécialisée et trois autres avec des instructions spécialisées. Une première version a été conçue avec une configuration de base sans multiplieur 16 bits, une seconde avec un multiplieur 16 bits et une troisième avec un MAC 16 bits. En ce qui concerne les versions avec des instructions spécialisées, les différentes variantes comprennent un, deux ou quatre MAC complexes.

Tableau 5.1 : Complexité matérielle des configurations de base du processeur Xtensa

Processeur		Fréquence (MHz)	Complexité matérielle du noyau (portes)
Xtensa de base sans multiplieur 16 bits	min	105	82240
	max	242	99090
Xtensa de base avec un multiplieur 16 bits	min	105	88010
	max	239	106000
Xtensa de base avec un MAC 16 bits	min	105	98010
	max	234	118100

En ce qui a trait à la complexité matérielle pour les processeurs avec des instructions spécialisées, seules les instructions spécialisées ont été synthétisées a priori pour obtenir une estimation de l'ajout matériel. Pour cette synthèse, la contrainte en termes de fréquence d'horloge a été fixée à 300 MHz, car le routage n'était pas pris en compte pour cette estimation. Cette contrainte a été fixée pour obtenir une performance satisfaisante après routage. Elle tient compte d'une dégradation attendue de l'ordre de 20% lors du placement-routage. Dans ce cas, l'objectif de 300 MHz avant placement-routage pour les

instructions spécialisées visait donc à obtenir des instructions spécialisée qui supportent la même fréquence que le noyau, à savoir 239 MHz. Pour estimer le nombre de portes pour chacune des instructions spécialisées, une équation tirée de la documentation de Tensilica [52] a été utilisée.

$$\text{Nombre de portes estimé} = \frac{\text{Aire totale des cellules}}{\text{Aire NAND gate 2-input drive 2X}} \quad (5\text{-}1)$$

$$\text{Nb total de portes} = \text{Nb de portes du processeur} + \text{Nb de portes pour les IS} \quad (5\text{-}2)$$

Voici le sommaire de la synthèse des instructions spécialisées (sans le noyau du processeur) pour les égaliseurs LTE-LMS et DFE-LMS. Cette synthèse ciblait une fréquence de 300 MHz.

Tableau 5.2 : Complexité matérielle des unités spécialisées

	Processeur	Aire totale	Nombre de portes estimé	Fréquence maximale
			Portes	MHz
LTE-LMS	TIE 1 MC	648101.44	31883	300
	TIE 2 MC	879365.19	43259	300
	TIE 4 MC	1261568.5	62061	300
DFE-LMS	TIE 1 MC	830142.88	40838	300
	TIE 2 MC	1020327.5	50194	300
	TIE 4 MC	1400046.62	68873	300
	Porte	Aire totale		
	NAND Gate 2-input Drive 2X	20.328		

5.1.2 Facteur d'accélération

Après avoir généré toutes les configurations, un profilage a été exécuté pour déterminer le nombre de cycles nécessaires pour effectuer la boucle de traitement. À l'aide de cette

information et de la fréquence d'horloge du processeur, le temps d'exécution d'une itération de la boucle a été déterminé. Connaissant le temps d'exécution d'une itération de la boucle, c'est-à-dire le temps pour la décision d'un symbole, il a été possible de déterminer le débit de traitement de l'égaliseur. L'équation suivante résume le raisonnement précédent.

$$F_{égaliseur} = \frac{1 \text{ bit}}{T_{exécution}} = \frac{1 \text{ bit}}{\text{Nb cycles x } T_{processeur}} = \frac{1 \text{ bit x } F_{processeur}}{\text{Nb cycles}} \quad \textbf{(5-3)}$$

Le tableau 5.3 effectue le sommaire des résultats pour la réalisation d'égaliseurs avec le Xtensa. La dernière colonne du tableau sera expliquée dans la prochaine section.

Tableau 5.3 : Sommaire des résultats pour l'implémentation des égaliseurs LTE-LMS et DFE-LMS avec le processeur Xtensa

	Processeur	Nombre de portes estimé	Nombre de cycles	Fréquence maximale	Temps d'exécution	Fréquence d'égalisation	Rapport AT
		Kportes		MHz	ns	Mbps	
LTE-LMS	Xtensa de base sans un multiplieur 16 bits	99.09	3990	242	16488	0.06	1
	Xtensa de base avec un multiplieur 16 bits	106.00	442	239	1850	0.54	8
	Xtensa de base avec un MAC 16 bits	118.10	434	234	1855	0.53	7
	TIE 1 MC 16 bits	137.88	55	239	231	4.32	51
	TIE 2 MC 16 bits	149.26	36	239	151	6.62	72
	TIE 4 MC 16 bits	168.06	28	239	118	8.47	82
DFE-LMS	Xtensa de base sans un multiplieur 16 bits	99.09	5650	242	23348	0.04	1
	Xtensa de base avec un multiplieur 16 bits	106.00	700	239	2929	0.34	7
	Xtensa de base avec un MAC 16 bits	118.10	686	234	2932	0.34	7
	TIE 1 MC 16 bits	146.84	72	239	302	3.31	52
	TIE 2 MC 16 bits	156.19	46	239	193	5.18	77
	TIE 4 MC 16 bits	174.87	33	239	139	7.19	95

En analysant ces données, il est possible de déterminer l'évolution de la complexité matérielle en fonction des performances de l'égaliseur. Cette analyse est reflétée dans la figure 5.6. Ce qui peut surprendre de ces résultats (voir le tableau 5.3), c'est que l'intégration d'un MAC au noyau du processeur ne permet pas d'accélération par rapport à un simple multiplieur, mais au contraire cela conduit à un ralentissement du traitement. Effectivement, l'exécution avec un MAC prend moins de cycle, mais impose une fréquence de traitement plus faible, ce qui a comme résultat un ralentissement. En fait, le gain en nombre de cycles économisés associé à l'intégration d'un MAC n'est pas assez important pour que cette option soit rentable.

Également, il est intéressant d'observer qu'il est possible d'obtenir un gain sur les performances à l'aide d'instructions spécialisées. Par rapport à un processeur Xtensa de base avec un multiplieur 16 bits, il est possible avec des instructions spécialisées contenant 4 multiplieurs complexes d'obtenir des gains en performance d'un facteur de 16 sur l'égaliseur LTE-LMS et d'un facteur de 21 pour le DFE-LMS.

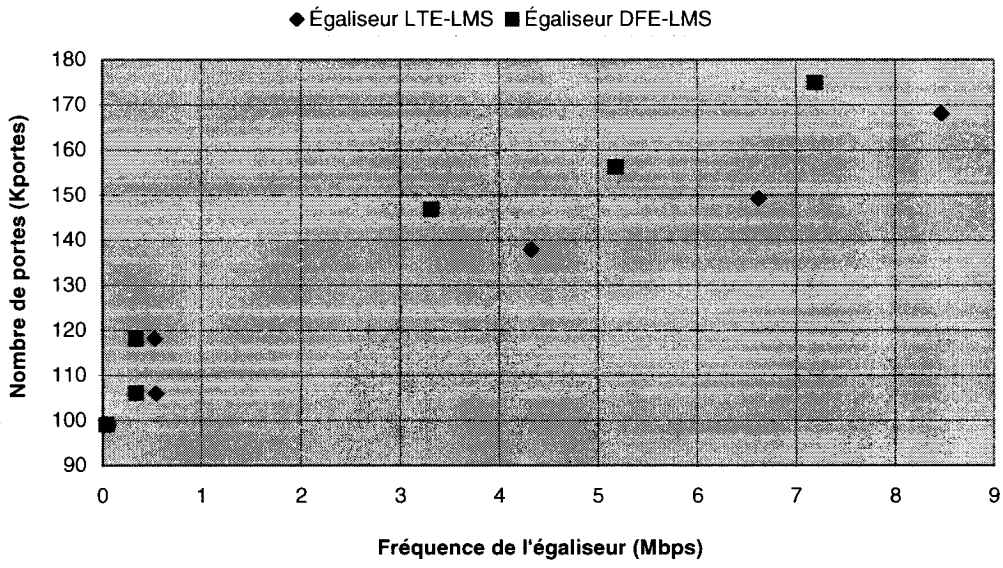

Figure 5.6 : Complexité matérielle en fonction de la fréquence de l'égaliseur pour le processeur Xtensa

5.1.3 Rapport de produits AT

Au-delà du constat très positif relatif aux intéressants gains de performance, il est pertinent de déterminer si ces gains sont plus importants que les coûts additionnels du matériel qu'il a fallu ajouter pour les obtenir. Le produit AT (Aire x Temps) reflète l'efficacité qui correspond aux options architecturales investiguées. Comme démontré dans le chapitre 3, cette métrique est l'équivalent de la métrique performance-aire (PA). Ainsi, si en doublant la complexité matérielle la période d'exécution de la boucle est diminuée de moitié, ceci se traduirait par un rapport de produit AT de 1. Une telle valeur démontre plutôt l'effet d'un compromis entre la complexité matérielle et la performance qu'un réel bénéfice en efficacité de calcul. Notons que ce compromis est souvent jugé avantageux car il permet d'ajuster le niveau de performance selon les besoins d'une application, de façon à rencontrer ces besoins à un coût minimal. Soulignons que dans l'évaluation du produit AT, dans le contexte présent, la variable T représente le temps d'exécution de la boucle d'égalisation et non la période de l'horloge. Nous allons nous intéresser dans ce qui suit à un rapport de produits AT qui compare l'efficacité relative de deux solutions possibles :

$$\text{Rapport } AT_i = \frac{A_1 T_1}{A_i T_i} \tag{5-4}$$

La figure 5.7 illustre le rapport AT_i en fonction des performances de différentes variantes de l'égaliseur. Le produit AT de référence est celui du processeur Xtensa sans multiplieur 16 bits. Cette métrique démontre que le gain en performance est considérablement plus important que l'ajout de matériel. De plus, le rapport AT_i semble saturer asymptotiquement lorsque les performances de l'égaliseur sont élevées. La tendance de la courbe semble démontrer qu'au fur et à mesure que du matériel est ajouté, le gain en performance devient moins important par rapport à cet ajout.

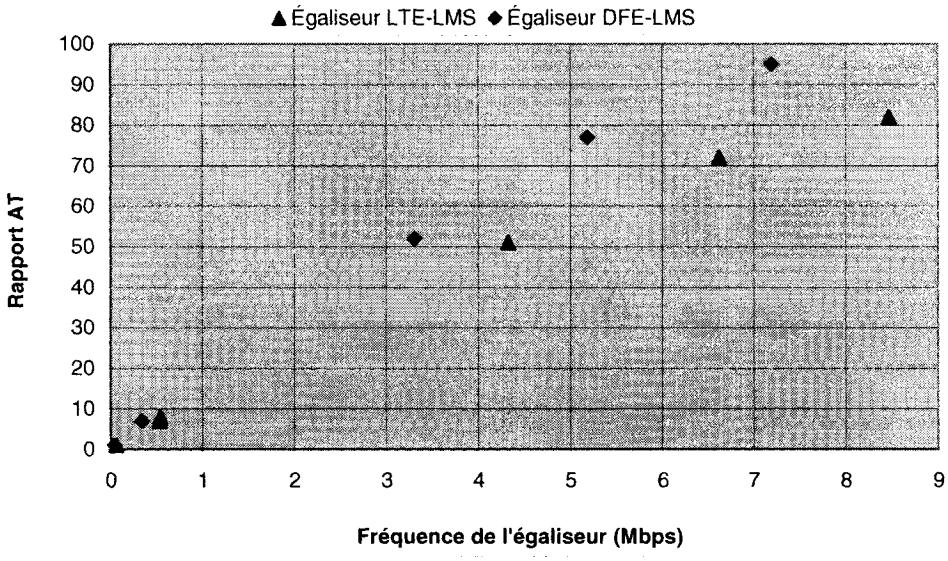

Figure 5.7 : Rapport AT_i en fonction de la fréquence de l'égaliseur pour le Xtensa

5.2 Implémentation avec le processeur Nios

En ce qui concerne l'implémentation des égaliseurs avec le processeur Nios, les mêmes instructions ont été introduites au sein du noyau. Comme avec le processeur Xtensa, les instructions spécialisées se résument à une unité spécialisée en parallèle avec l'ALU. Dans le cadre du processeur Nios, le nombre d'instructions spécialisées est limité à 5. Toutes les instructions spécialisées prennent 2 entrées de 32 bits, quelques signaux de contrôle, une entrée de 11 bits (préfixe) optionnelle et fournissent à la sortie un résultat de 32 bits. L'utilité de l'entrée optionnelle de 11 bits est déterminée par le concepteur. Ainsi, il est possible d'utiliser ces 11 bits pour définir 2048 (2^{11}) instructions différentes. Cependant, l'utilisation de cette entrée de 11 bits nécessite le chargement du préfixe, ce qui a pour conséquence un coût d'un cycle d'exécution. C'est cette technique qui a été utilisée dans le cadre de cette implémentation. La figure 5.8 illustre l'interface de l'unité spécialisée.

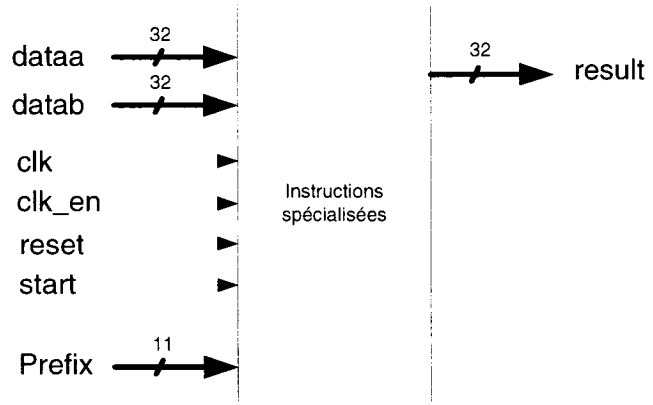

Figure 5.8 : Interface de l'unité spécialisée dans le processeur Nios

Ainsi, il n'est pas possible de passer plus que deux paramètres à l'unité spécialisée par appel, car le préfixe de 11 bits est réservé à la définition de l'instruction. Ainsi pour dédier les variables redondantes à des registres spécialisés, il faut par conséquent intégrer des registres à l'unité spécialisée. À ces registres, il faut ajouter des instructions de lecture et d'écriture. De cette façon, il sera possible que toutes les autres instructions possèdent plus de deux opérandes, soit les entrées et les registres intégrés dans l'unité spécialisée.

Toutes les instructions spécialisées ont été implémentées pour être exécutées en 3 cycles parce que leur chemin critique le nécessitait. Contrairement à la conception avec le processeur Xtensa, il n'a pas été possible de conserver la fréquence maximale du processeur de base pour les processeurs comportant des instructions spécialisés (115 MHz). Ainsi, les processeurs avec une unité spécialisée ne sont pas capables d'opérer à plus de 100 MHz. Ce ralentissement est probablement dû au routage des interconnexions entre le noyau et l'unité spécialisée. Il est à spécifier que le noyau utilise peu ou pas d'éléments DSP du FPGA. Cependant, les unités spécialisées utilisent de façon considérable ces éléments DSP qui se retrouvent à des endroits fixes dans le FPGA. Ainsi, le routage entre les éléments de l'unité spécialisée et les éléments DSP peut

occasionner des délais pour la propagation des signaux. Tous les égaliseurs réalisés à l'aide du processeur Nios ont été implémentés et validés sur une plate-forme FPGA.

5.2.1 Complexité matérielle

Comme dans l'analyse avec le processeur Xtensa, six différentes versions du processeur ont été utilisées pour chaque égaliseur. Trois versions ont été réalisées avec un processeur Nios sans instruction spécialisée et trois autres avec des instructions spécialisées. Une première version a été conçue avec une configuration de base sans multiplieur 16 bits (multiplication logicielle), une seconde avec une unité fonctionnelle qui effectue une multiplication partielle (MSTEP) et une autre avec un multiplieur 16 bits. En ce qui concerne les versions avec des instructions spécialisées, les différentes variantes comprennent un, deux ou quatre multiplieurs complexes.

La complexité matérielle de ces processeurs a été obtenue après la synthèse pour un FPGA d'Altera (Stratix EP1S40). Le tableau 5.4 effectue le sommaire des résultats pour la complexité matérielle des processeurs. Il est à noter que l'ajout d'un multiplieur à la configuration de base augmente peu la complexité matérielle (Nombre de LC). Par contre, lorsqu'une unité spécialisée est introduite dans le processeur, la complexité matérielle double et la fréquence d'horloge diminue. Le nombre de LC (« *logic cells* ») est considéré comme indice de la complexité matérielle. Cependant, si seul le nombre de LC était considéré, les résultats seraient un peu biaisés, car ceux-ci ne tiennent pas compte du nombre d'éléments DSP du FPGA. Ces éléments DSP représentent une certaine complexité matérielle qu'il n'est pas possible de soustraire de l'analyse. Par conséquent, il a fallu évaluer la complexité matérielle des éléments DSP en termes de LC. Les éléments DSP sont utilisés pour effectuer des multiplications 16x16. Ainsi, pour chaque multiplication de 16 bits, deux éléments DSP sont intégrés à l'unité spécialisée. Après la synthèse d'un multiplieur de 16 bits sans élément DSP, la complexité matérielle d'un élément DSP a été évaluée à 200 LC (400 LC/ 2 éléments DSP). Ainsi, ce facteur de correction a été tenu en compte dans l'analyse.

Complexité matérielle=Nb LC + Nb d'éléments DSP x (200 LC/élément DSP) (5-5)

Dans le tableau 5.4, la colonne étiquetée « LC » exhibe le nombre total de LC pour l'implémentation des différentes configurations. Ce nombre ne comprend pas le facteur de corrections pour les éléments DSP. La colonne étiquetée « registres » comprend le nombre total de registres nécessaire pour l'implémentation. Une cellule logique (ou LC) est constituée principalement de deux sections, soit une LUT et un registre. Par conséquent, une cellule logique peut être utilisée seulement pour la LUT, seulement pour le registre ou pour les deux. Cette information est comprise dans les trois colonnes étiquetées « LUT seul LC», « Registre-seul LC » et « LUT/Registre LC ».

Tableau 5.4 : Complexité matérielle des configurations du processeur Nios

	Processeur	Fréquence maximale	LC	LUT-seul LC	Registre-seul LC	LUT/Registre LC	Registres	Éléments DSP
		MHz	u	u	u	u	u	u
LTE-LMS	Base avec un multiplieur logiciel	115	2772	1605	75	1092	1167	0
	Base avec un multiplieur MSTEP	115	2797	1623	69	1105	1174	0
	Base avec un multiplieur 16 bits	115	2824	1653	68	1103	1171	2
	Unité spécialisée avec 1 MC	100	5730	3546	91	2093	2184	14
	Unité spécialisée avec 2 MC	100	6057	3597	79	2381	2460	22
	Unité spécialisée avec 4 MC	100	6591	3645	84	2862	2946	38
DFE-LMS	Base avec un multiplieur logiciel	115	2772	1605	75	1092	1167	0
	Base avec un multiplieur MSTEP	115	2797	1623	69	1105	1174	0
	Base avec un multiplieur 16 bits	115	2824	1653	68	1103	1171	2
	Unité spécialisée avec 1 MC	100	6674	4008	63	2603	2666	14
	Unité spécialisée avec 2 MC	100	7714	4240	138	3336	3474	22
	Unité spécialisée avec 4 MC	100	7818	4341	149	3328	3477	38

5.2.2 Facteur d'accélération

En ce qui concerne le profilage avec le Nios, la technique utilisée est basée sur un compteur de temps (« *timer* »). En fait, le concept consiste à prendre en note la valeur du

compteur au début et à la fin de l'exécution du code à profiler. La différence entre les deux valeurs donne un résultat auquel il faut soustraire la surcharge relative au profilage pour obtenir le nombre de cycles d'exécution. Afin d'éliminer les cycles d'exécution de surcharge relatifs au profilage, il s'agit de profiler un segment de code vide et de soustraire les deux valeurs afin de déterminer le nombre de cycles de surcharge pour le profilage. De plus, le profilage a été exécuté sur plusieurs itérations de l'égaliseur. Avec le profilage de plusieurs itérations, il suffit à présent de diviser le nombre de cycles d'exécution par le nombre d'itération pour obtenir le temps d'exécution d'une itération. Pour le profilage de plusieurs itérations, il s'agit de s'assurer que le compteur ne retourne pas à sa valeur initiale pendant le temps du profilage.

Tableau 5.5 : Sommaire des résultats pour l'implémentation des égaliseurs LTE-LMS et DFE-LMS avec le processeur Nios

	Processeur	Fréquence maximale	LC	Nombre de cycles	Temps d'exécution	Fréquence d'égalisation	Rapport AT
		MHz	u	u	ns	Mbps	
LTE-LMS	Base avec un multiplieur logiciel	115	2772	19063	165766	0.006	1
	Base avec un multiplieur MSTEP	115	2797	9889	85992	0.011	2
	Base avec un multiplieur 16 bits	115	3224	3619	31470	0.031	4
	Unité spécialisée avec 1 MC	100	8530	148	1480	0.675	37
	Unité spécialisée avec 2 MC	100	10457	84	840	1.19	53
	Unité spécialisée avec 4 MC	100	14191	66	660	1.515	49
DFE-LMS	Base avec un multiplieur logiciel	115	2772	28960	251827	0.003	1
	Base avec un multiplieur MSTEP	115	2797	15060	130957	0.007	2
	Base avec un multiplieur 16 bits	115	3224	5560	48348	0.02	6
	Unité spécialisée avec 1 MC	100	9474	192	1920	0.52	51
	Unité spécialisée avec 2 MC	100	12114	109	1090	0.917	70
	Unité spécialisée avec 4 MC	100	15418	75	750	1.333	80

Le tableau 5.5 effectue le sommaire des résultats obtenus pour les performances avec le processeur Nios. Le nombre de LC comprend le facteur de correction pour les éléments DSP discuté précédemment. Il est possible d'atteindre un gain d'un facteur de 47.7 (31470/660) en performance pour l'égaliseur LTE-LMS et de 64.5 (48348/750) pour l'égaliseur DFE-LMS par rapport à une configuration de base avec un multiplicateur 16 bits. Si les exécutions en nombre de cycles pour le processeur Nios et le processeur Xtensa sont comparées, il est remarquable que celles pour le Nios soient plus de deux fois plus élevées que celles du Xtensa. L'explication principale à cette différence est que le processeur Nios n'effectue aucune autre instruction en parallèle lorsque celui-ci effectue une instruction spécialisée, ce qui n'est pas le cas du Xtensa. Ainsi, le pipeline est bloqué jusqu'à la fin de l'exécution de l'instruction spécialisée, c'est-à-dire 2 cycles pour cette unité spécialisée. C'est à partir du code au niveau assembleur et du profilage du code que cette conclusion a été tirée.

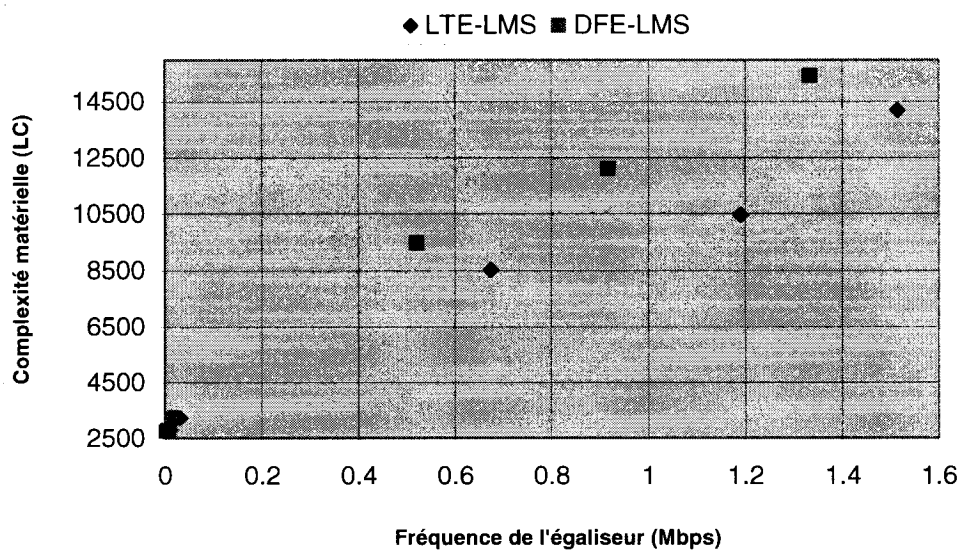

Figure 5.9 : Complexité matérielle en fonction de la fréquence de l'égaliseur pour le processeur Nios

Pour ce qui est de la complexité matérielle en fonction des performances de l'égaliseur, celle-ci semble évoluer comme pour le processeur Xtensa. Ce ne serait pas nécessairement le cas si le facteur de correction pour les éléments DSP n'avait pas été tenu compte pour la complexité matérielle. Comme la figure 5.9 le démontre, la complexité matérielle croît linéairement en fonction de la fréquence de l'égaliseur. Intuitivement, la courbe de la complexité matérielle devrait croître rapidement. En fait, l'efficacité de l'égaliseur mesurée par un rapport de produits AT devrait plafonner et même régresser si la complexité matérielle augmente plus vite que les performances.

5.2.3 Rapport de produits AT

La figure 5.10 illustre le rapport de produits AT en fonction de la fréquence de l'égaliseur. Le produit AT de référence est celui du processeur Nios avec multiplicateur logiciel. Comme il est possible de le constater, la forme de la courbe ressemble fortement à celle obtenue avec le Xtensa. Cela démontre que les résultats obtenus sont en accord avec ceux obtenus avec le Xtensa et que ceux-ci sont probablement représentatifs de tous les autres processeurs configurables. Comme dans le cas du processeur Xtensa, cette métrique montre que le gain en performance est considérablement plus important que l'ajout de matériel.

Également, ce qui est intéressant dans la courbe de la figure 5.10, c'est qu'elle finit par décroître pour l'égaliseur LTE-LMS. Ce phénomène n'est toutefois pas observable dans la courbe du Xtensa. Cela signifie qu'à ce stade, l'augmentation de la complexité matérielle est plus rapide que celle des performances. Cette observation vient confirmer notre intuition face à l'évolution de la complexité matérielle en fonction des performances. De plus, cette courbe suggère qu'il y ait un point où l'augmentation des performances est équivalente ou supérieure à l'augmentation de la complexité matérielle (pente nulle ou négative).

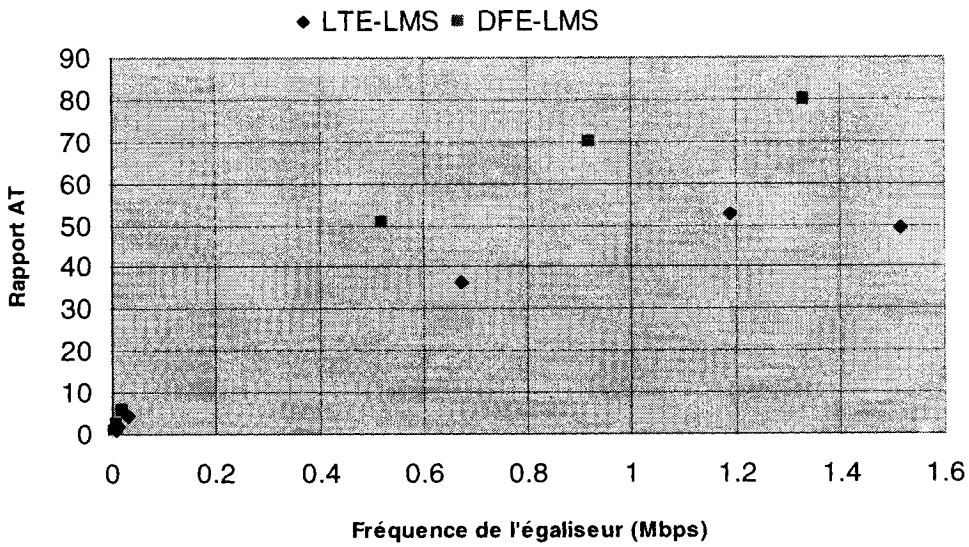

Figure 5.10 : Rapport AT_i en fonction de la fréquence de l'égaliseur pour le processeur Nios

CHAPITRE 6
CONCLUSION

Le présent chapitre se consacre en premier lieu à la synthèse des quatre chapitres précédents. Il sera discuté par la suite des limitations des travaux effectués, puis des améliorations possibles qui pourraient êtres apportées. Pour finir, des indications pour des recherches futures seront dévoilées.

6.1 Synthèse des travaux

La première phase de ce projet consistait à explorer la technologie de processeur configurable. Ainsi, il a été possible d'évaluer quels seraient les paramètres sur lesquels il était possible d'influer pour optimiser des processeurs configurables. Les technologies utilisées, soit le processeur Xtensa et le processeur Nios, sont des processeurs configurables avec un noyau fixe et paramétrable. Avec de tels processeurs, il est, comme discuté dans le chapitre 2, possible d'intégrer certaines unités fonctionnelles, comme des multiplications, divisions et des unités à virgule flottante. De plus, il est souvent possible de choisir le nombre de registres physiques, la taille de l'antémémoire et son type. Bien entendu, ce qui fait la force d'une telle technologie est la possibilité d'intégrer une unité spécialisée dans le processeur. À part les technologies utilisées, il existe d'autres technologies avec lesquelles le noyau du processeur est flexible, i.e. pour lesquelles l'élaboration de l'architecture du noyau est laissée au soin du concepteur. Parmi celles-ci, il est possible de retrouver la technologie Lisatek et Target Compiler.

Dans le chapitre 3, les techniques qui permettent une amélioration à l'aide d'instructions spécialisées sont énumérées. Parmi ces techniques, les registres spécialisés sont un moyen d'augmenter la localité des variables. Pour sa part, la fusion d'instructions simples permet d'ajouter une instruction pour une fonctionnalité qui se réalise mieux en matériel

qu'en logiciel. En ce qui concerne les opérations vectorielles, elles permettent d'effectuer plusieurs opérations du même type sur un ensemble de données. La parallélisation sous forme VLIW permet d'effectuer plusieurs opérations différentes dans la même période. La technique du pipelinage consiste à étaler l'exécution d'une instruction avec un long chemin critique sur plusieurs cycles, ce qui contribue à conserver la fréquence d'horloge de la configuration de base. Comme dernière technique, le partage des unités opératives permet de réduire la complexité matérielle reliée à l'introduction d'une unité spécialisée.

De plus, devant plusieurs d'instructions spécialisées, il faut être en mesure de déterminer quelles sont celles qui vont permettre d'atteindre les objectifs. Un ensemble de métrique de qualité a été développé afin d'évaluer la pertinence d'une instruction spécialisée face à l'objectif poursuivi. Parmi celles-ci, il y a la métrique de performance (P), celle de performance-aire (PA) et celle de performance-puissance (PP). La première métrique caractérise une instruction spécialisée pour un simple objectif de gain de performances. La seconde métrique, soit celle de performance-aire, caractérise une instruction pour l'accélération qu'elle procure en fonction de l'aire qu'elle occupe. Cette métrique est pertinente dans un contexte d'accélération sous une contrainte d'aire. La dernière métrique, soit celle de la performance-puissance, sert à qualifier une instruction en fonction de la réduction de puissance qu'elle procure.

Finalement, les limites de l'accélération ont été explorées. Comme premier point soulevé, la loi d'Amdhal donne un indice sur l'accélération globale atteignable en fonction de la portion accélérable de l'application. Parmi les limites de l'accélération, il faut prendre en considération les contraintes physiques de la technologie, soit le nombre permis d'entrées/sorties des IS, le nombre total d'IS, l'aire alloué pour une unité spécialisée, ainsi que la bande passante mémoire. De plus, l'évolution du gain local pour une IS en fonction du chargement des données a été démontrée. Cette évolution démontre que plus le temps de chargement des données est élevé par rapport à l'exécution de l'IS, moins le matériel ajouté est efficace. En d'autres mots, le gain local diminue à une fraction du gain effectif lorsque le temps de chargement des données augmente. Il faut par conséquent,

avant d'ajouter des IS, faire en sorte que les variables soient locales ou que le temps de chargement soit court face à celui de l'exécution. En dernier lieu, la parallélisation des opérateurs dans un pipeline a été analysée. Ce qui ressort de cette analyse, c'est que le gain local évolue en pallier et celui-ci est maximal lorsque le niveau de parallélisation est un facteur du nombre d'opérations à effectuer. De plus, si les entrées sont consécutives dans le pipeline, le gain local est augmenté par un facteur plus petit que 2 à chaque fois que le niveau de parallélisation est doublé (sauf pour un seul étage de pipeline, n=1). Cependant, si les entrées ne sont pas consécutives dans le pipeline, le gain local évolue linéairement avec le niveau de parallélisation.

Dans le chapitre 4, une méthodologie de conception pour les processeurs configurables à noyau fixe et paramétrable a été présentée. Celle-ci se divise principalement en quatre phases, soit celle du développement logiciel, de l'exploration architecturale, de la génération d'une unité spécialisée et celle de l'implémentation. La première phase consiste à développer l'application sous un langage comme le C/C++ et d'effectuer des optimisations logicielles. La seconde phase a comme objectif de trouver les paramètres optimaux du noyau et d'évaluer les performances pour la configuration de base. La troisième vise à exploiter les techniques d'optimisation du chapitre 3 pour générer une unité spécialisée. La génération d'IS peut être réalisée de façon manuelle ou automatique. En ce qui concerne la génération automatique, celle-ci donne de plus faibles accélérations. Ce phénomène peut s'expliquer par le fait que les algorithmes de génération automatique d'IS s'attaquent à des sections de code délimitées par des accès mémoire et des transferts de contrôle, alors que ces limites peuvent être déplacées. Finalement, la phase d'implémentation consiste à passer du modèle du processeur à un prototype. La suggestion, pour cette phase, est de maintenir le chemin critique du processeur hors de l'unité spécialisée en réalisant un pipeline sur les longues IS. Cette recommandation a pour effet d'éviter de ralentir une section du code pour en accélérer une autre.

Dans le chapitre 5, les résultats de l'implémentation ont été exposés. En somme, l'ajout d'instructions spécialisées à la configuration de base d'un processeur, pour les deux technologies, permet d'obtenir des gains considérables pour un algorithme d'égalisation. Dans un premier temps, une mise en œuvre a été effectuée avec le processeur Xtensa qui visait une réalisation ASIC. Avec un égaliseur LTE-LMS, il est possible d'obtenir un gain de 16 sur les performances par rapport à une configuration de base avec un multiplieur 16 bits et avec un égaliseur DFE-LMS, un gain de 21. Dans un deuxième temps, une réalisation à l'aide du processeur Nios a été accomplie, ce qui ciblait une plate-forme FPGA. Avec une telle technologie, il est possible d'atteindre un gain de 47.7 pour l'égaliseur LTE-LMS et un gain de 64.5 pour le DFE-LMS. Bien qu'il soit possible d'obtenir d'importants gains avec des instructions spécialisées, les débits de traitement atteignables dans un contexte d'égalisation ne sont pas très intéressants. Il faudrait disposer de plus d'un processeur en parallèle afin d'atteindre des débits satisfaisants pour l'égalisation dans des systèmes de communication performants.

De plus, les résultats démontrent que la complexité matérielle évolue quasi-linéairement avec la fréquence de l'égaliseur. Cependant, après l'atteinte d'un certain niveau d'accélération, il est attendu que la complexité matérielle augmenterait considérablement par rapport aux performances. Le rapport de produits AT démontre que le gain en performance peut, avant un point de saturation, être plus important que l'ajout de matériel. Cependant, l'importance du gain des performances par rapport à l'ajout de matériel tend à diminuer avec l'augmentation des performances de l'égaliseur. De plus, il est observable avec les résultats du processeur Nios que la courbe du rapport de produits AT finit par décroître. Cela signifie qu'à un certain stade, l'augmentation de la complexité matérielle est plus élevée que celle des performances. Il est à remarquer que l'évolution des performances en fonction de la complexité matérielle évolue de la même façon pour les deux technologies, ce qui porte à croire que cette tendance est maintenue avec d'autres technologies.

6.2 Limitations des travaux

Les données recueillies au cours de ce travail concernaient principalement des applications de traitement de signal. Ces applications sont axées sur des flots de données et les opérations à effectuer sont relativement semblables, ce qui se prête bien à l'accélération. Cependant, les résultats obtenus ne seraient probablement pas tout à fait les mêmes si l'application était différente. Bien entendu, des applications avec beaucoup de transfert de contrôle auraient de moins bons facteurs d'accélération. En somme, le domaine d'application de ces travaux est restreint. Cependant, il y a de bonnes raisons de croire que ces résultats se refléteraient dans d'autres domaines d'application.

Les deux technologies employées durant la réalisation de ces travaux, soit le processeur Xtensa et le processeur Nios, étaient des processeurs à noyaux fixes et paramétrables. Ainsi, il n'était pas possible de modifier le noyau du processeur comme avec des technologies comme Lisatek et Target Compiler. Ainsi, ces travaux se limitent à un noyau de type RISC pipeliné sur 5 étages. Par contre, une augmentation ou une réduction du nombre d'étages du pipeline aurait peut-être été avantageux dans un tel contexte. D'autres aspects du processeur auraient pu être étudiés comme des architectures superscalaires.

De plus, la génération d'IS a été réalisée manuellement. Par contre, il existe des outils automatiques sur le marché actuel. Bien que les résultats publiés démontrent de faibles accélérations avec de tel outil, il aurait été intéressant de connaître les résultats avec cette application. Avec de tels résultats, il aurait été possible d'effectuer une comparaison entre la méthode manuelle et automatique.

6.3 Indications de recherches futures

La méthodologie de conception avec des processeurs configurables utilisent un ensemble d'outils qui permettent d'automatiser certaines étapes comme la génération d'outils de développement logiciel, la génération de descriptions matérielles de processeurs et bien d'autres. Cependant, il serait possible d'automatiser davantage cette méthodologie afin d'obtenir de meilleurs résultats. Par exemple, la phase d'exploration architecturale du processeur est réalisée manuellement, i.e. les options sont sélectionnées manuellement par le concepteur. Il est évident qu'un outil automatique pour l'accomplissement de cette phase serait bénéfique. Un tel outil pourrait, à partir du programme de l'application, sélectionner automatiquement les unités fonctionnelles nécessaires parmi celles qui sont disponibles, le nombre de registres physiques adéquat, la taille et le type d'antémémoire optimaux. De plus, il serait intéressant d'effectuer de la recherche pour perfectionner les outils automatisés actuels afin de les rendre plus performants comme dans le cas de la génération d'IS.

Les résultats précédemment discutés concernent l'utilisation d'un seul processeur. Cependant, les circuits actuels utilisent un ensemble de processeurs pour réaliser leurs fonctionnalités. Des travaux futurs pourraient être réalisés sur le partitionnement de fonctionnalités sur plusieurs processeurs et d'accélérer ces différents processeurs à l'aide d'unités spécialisées.

Afin de vérifier le fonctionnement d'un algorithme, il faut développer en premier lieu un modèle mathématique. Par la suite, il faut quantifier les différentes opérandes sur un certain nombre de bits (virgule fixe) afin de se rapprocher d'une implémentation réelle. Dans le cadre de ce travail, les données ont été quantifiées sur un certain nombre de bits qui semblait donner de bons résultats. Cependant, la quantification de ces données n'est pas nécessairement optimale. Des travaux réalisés au GRM [9] ont conduit au développement d'un outil qui permet d'optimiser la quantification des opérateurs dans un

tel contexte. Il aurait été intéressant d'intégrer cet outil dans le flot de conception afin d'obtenir une meilleure quantification.

BIBLIOGRAPHIE

[1] ABU-AL-SAUD, W.A., STUBER, G.L., « *Modified CIC filter for sample rate conversion in software radio systems* », IEEE Signal Processing Letters, vol. 10 p.152-154, Mai 2003.

[2] ALTERA, www.altera.com/literature/lit-nio.jsp.

[3] ARC COMPANY, www.arc.com.

[4] ARNOLD, M., CORPORAAL, H., « *Designing domain-specific processors* », Proceedings of the ninth international symposium on Hardware/software codesign, p.61-66, Avril 2001.

[5] ATASU, K., POZZI, L., IENNE, P., « *Automatic application-specific instruction-set extensions under microarchitectural constraints* », International Journal of Parallel Programming, Volume 31, Décembre 2003.

[6] BAJOT, Y., MEHREZ, H., « *Customizable DSP architecture for ASIP core design* », Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on, Volume 4, p.302-305, 6-9 Mai 2001.

[7] BRISK, P., KAPLAN, A., KASTNER, R., SARRAFZADEH, M., « *system synthesis: Instruction generation and regularity extraction for reconfigurable processors* », Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems, p.262-269, Octobre 2002.

[8] BURACCHINI, E., « *The software radio concept* », IEEE Communications Magazine, p.138 –143, Septembre 2000.

[9] CANTIN, M.-A., SAVARIA, YVON, « *An Automatic Word Length Determination Method* », WSEAS Transactions on Information Science and Applications, vol.1, p.1440-1448, 2004.

[10] CHEUNG, N., HENKEL, J., PARAMESWARAN, S., « *Rapid configuration and instruction selection for an ASIP: a case study* », Design, Automation and Test in Europe Conference and Exhibition, p.802–807, 2003.

[11] CLARK, N., ZHONG, H., MAHLKE, S., « *Processor Acceleration Through Automated Instruction Set Customization* », Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, Decembre 2003.

[12] CRITICAL BLUE, www.criticalblue.com.

[13] CUMMINGS, M., HARUYAMA, S., « *FPGA in the software radio* », Communications Magazine, IEEE, vol. 37, p.108–112, Février 1999.

[14] DICK, C.H., PEDERSEN, H.M., « *Implementation of FPGA signal processing datapaths for software defined radios* », Proceeding Conference, Communication Design China, p.241-247, Août 2001.

[15] EFSTATHIOU, D., FRIDMAN, L., ZVONAR, Z., « *Recent developments in enabling technologies for software defined radio* », IEEE Communications Magazine, vol. 37, p.112–117, Août 1999.

[16] GIRAU, G., MARTINA, M., MOLINO, A., TERRENO, A., VACCA, F., « *FPGA digital down converter IP for SDR terminals* », Signals, Systems and Computers, 2002. Conference Record of the Thirty-Sixth Asilomar Conference on, vol. 2, p.1010-1014, 3-6 Novembre 2002.

[17] GOODWIN, D., PETKOV, D., « *Automatic generation of application specific processors* », Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems, Octobre 2003.

[18] GRAY, A.A., HOY, S.D., GHUMAN, P., « *Parallel VLSI equalizer architectures for multi-Gbps satellite communications* », Global Telecommunications Conference, IEEE, vol. 1, p.315-319, 25-29 Novembre 2001.

[19] GSCHWIND, M., « *Instruction set selection for ASIP design* », Hardware/Software Codesign, 1999. (CODES '99) Proceedings of the Seventh International Workshop on, p.7-11, 3-5 Mai 1999.

[20] GUNN, J.E., BARRON, K.S., RUCZCZYK, W., « *A low-power DSP core-based software radio architecture* », IEEE Journal on selected areas in communications, p. 574 –590, Avril 1999.

[21] HAGHIGHAT, A., « *A review on essentials and technical challenges of software defined radio* », Proceeding MILCOM 2002, vol. 1, p.377-382, 7-10 Octobre 2002.

[22] HENNESSY, J., PATTERSON, D.A., « *Computer Architecture : A quantitative approach* », Morgan Kaufmann Publishers, 3^{e} édition, 2003.

[23] HOFFMANN, A., FIEDLER, F., NOHL, A., PARUPALLI, S., « *A Methodology and Tooling Enabling Application Specific Processor Design* », VLSI Design, 2005. 18th International Conference on, p.399–404, 3-7 Janvier 2005.

[24] HOFFMANN, A., SCHLIEBUSCH, O., NOHL, A., BRAUN, G., WAHLEN, O., MEYR, H., « *A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA* », Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM International Conference on, p.625–630, 4-8 Novembre 2001.

[25] HUANG, X.H., DU, K.-L., LAI, A.K.Y., CHENG, K.K.M., « *A unified software radio architecture* », IEEE Third Workshop on Signal Processing Advances in Wireless Communications, p.330-333, 20-23 Mars 2001.

[26] IMPROV SYSTEMS, INC. , www.improvsys.com.

[27] ISHII, H., KAWAMURA, S., SUZUKI, T., KURODA, M., HOSOYA, H., FUJISHIMA, H., « *An adaptive receiver based on software defined radio techniques* », The 12th IEEE International Symposium on personal, indoor and

mobile radio communications, vol. 2, p. G-120 - G-124, 30 Septembre-3 Octobre 2001.

[28] JAIN, M.K., BALAKRISHNAN, M.K., « *ASIP design methodologies: survey and issues* », A.;VLSI Design, 2001. Fourteenth International Conference on, p.76-81, 3-7 Janvier 2001.

[29] KASTENS, U., LE, D.K., SLOWIK, A., THIES, M., « *Feedback driven instruction-set extension* », ACM SIGPLAN Notices , Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools, vol. 39, p.126-135, Juin 2004.

[30] KEUTZER, K., MALIK, S., NEWTON, A.R., « *From ASIC to ASIP: the next design discontinuity* » Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on, p.84-90, 16-18 Septembre 2002.

[31] KOKOZINSKI, R., GREIFENDORF, D., STAMMEN, J., JUNG, P., « *The evolution of hardware platforms for mobile Software Defined Radio Terminals* », The 13th IEEE International Symposium on personal, indoor and mobile radio communications, vol. 5, p.2389-2393, Septembre 2002.

[32] KUCUKCAKAR, K., « *An ASIP design methodology for embedded systems* », Hardware/Software Codesign, 1999. (CODES '99) Proceedings of the Seventh International Workshop on, p.17–21, 3-5 Mai 1999.

[33] LAVIGUEUR, B., « *Multitraitement et processeurs configurables sur une plate-forme de haut niveau* », Mémoire de maîtrise, École Polytechnique de Montréal, 2004.

[34] MACLEOD, J.R., Nesimoglu, T., Beach, M.A., Warr, P.A., « *Enabling technologies for software defined radio transceivers* », Proceeding MILCOM 2002, vol. 1, p. 354-358, 7-10 Octobre 2002.

[35] MIRANDA, H.C., PINTO, P.C., SILVA, S.B., « *A self-reconfigurable receiver architecture for software radio systems* », Radio and Wireless Conference RAWCON '03, 10-13 Août 2003.

[36] MITOLA, J., « *Software Radio Architecture* », John Wiley & Sons, 543 pages, 2000.

[37] MITOLA, J., « *The software radio architecture* », IEEE Communications Magazine, vol. 33, p. 26–38, Mai 1995.

[38] ORTIZ, S., « *Software radios add flexibility to wireless technology* », Computer, vol. 36, p. 23-25, Août 2003.

[39] PAN YU, TULIKA MITRA, « *Application specific processors: Scalable custom instructions identification for instruction-set extensible processors* », Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, p.69-78, Septembre 2004.

[40] PAN YU, TULIKA MITRA, « *Design methodologies for ASIP: Characterizing embedded applications for instruction-set extensible processors* », Proceedings of the 41st annual conference on Design automation, p.723-728, Juin 2004.

[41] PEES, S., HOFFMAN, A., ZIVOJNOVIC, V., MEYR, H., « *LISA-machine description language for cycle-accurate models of programmable DSP architectures* », Proceedings of the 36th ACM/IEEE conference on Design automation, Juin 1999.

[42] PEYMANDOUST, A., POZZI, L., IENNE, P., DE MICHELI, G., « *Automatic instruction set extension and utilization for embedded processors* », Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on, p.108-118, 24-26 Juin 2003.

[43] PROAKIS, J.G., « *Digital Communications* », quatrième édition, McGraw-Hill Series in Electrical and Computer Engineering, 2001, ISBN 0-07-232111-3.

[44] QUINN, D., LAVIGUEUR, B., BOIS, G., ABOULHAMID, E.M., « *A System Level Exploration Platform and Methodology for Network Applications Based on Configurable Processors* », Design and Test in Europe, 2004.

[45] SALKINTZIS, A.K., HONG NIE, MATHIOPOULOS, P.T, « *ADC and DSP challenges in the development of software radio base stations* », IEEE Personal Communications, vol. 6, p.47-55, Août 1999.

[46] SCHLIEBUSCH, O., HOFFMANN, A., NOHL, A., BRAUN B., MEYR H., « *Architecture Implementation Using the Machine Description Language LISA* », Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design, Janvier 2002.

[47] SDR FORUM, www.sdrforum.org.

[48] SHEKHAR, C., RAJ SINGH, MANDAL, A.S., BOSE, S.C., SAINI, R., TANWAR, P., « *Application Specific Instruction Set Processors: redefining hardware-software boundary* », VLSI Design, 2004. Proceedings. 17th International Conference on, p.915-918, 2004,

[49] SRIKANTESWARA, S., « *Design and Implementation of a Soft Radio Architecture for Reconfigurable Platforms* », Doctor Thesis, Virginia Polytechnic, 2001.

[50] TANGUAY, B., SAVARIA, Y., SAWAN, M., « *Accelerating equalization algorithms using the Xtensa configurable processor* », ICM 2004 Proceedings. The 16th International Conference on, p.434-437, 6-8 Décembre 2004.

[51] TARGET COMPILER TECHNOLOGIES, www.retarget.com.

[52] TENSILICA, « *Xtensa Microprocessor Overview Handbook* », Août 2002.

[53] WANG, A., KILLIAN, E., MAYDAN, D., ROWEN, C., « *Hardware/software instruction set configurability for system-on-chip processors* », Design Automation Conference, 2001. Proceedings, p.184-188, 18-22 Juin 2001.

[54] WIESLER, A, JONDRAL, F.K., « *A Software Radio for Second- and Third-Generation Mobile systems* », IEEE transactions on vehicular technology, p.738 –748, Juillet 2002.

[55] YOSHIDA, H., OTAKA, S., KATO, T., TSURUMI, H., « *A software defined radio receiver using the direct conversion principle: implementation and evaluation* », The 11th IEEE International Symposium on personal, indoor and mobile radio communications, vol. 2, p.1044-1048, Septembre 2000.

ANNEXES

ANNEXE A ALGORITHMES D'ÉGALISATION

Lors de la réception de données en bande de base, une distorsion causée par le canal de télécommunication est toujours présente sur le signal. Cette distorsion provient de l'interférence inter-symbole et de la propagation multi-chemin. Pour se défaire de cette distorsion, le signal doit être filtré avec l'inverse de la fonction de transfert du canal. Afin d'ajuster les coefficients du filtre à une telle fonction de transfert, plusieurs algorithmes de mise à jour des coefficients existent, dont le RLS (« Recursive Least-Squares ») et le LMS (« Least Means Square ») [43]. L'algorithme le plus populaire est le LMS étant donné sa simplicité de mise en œuvre.

Dans le cadre de l'algorithme LMS, l'objectif premier est de réduire l'erreur quadratique moyenne. La première étape consiste à effectuer un filtrage à l'aide d'un FIR avec des coefficients complexes W(n) et des données complexes X(n). Ces deux vecteurs sont des vecteurs colonnes de longueur N, où N représente le nombre de coefficients du filtre. Par la suite, une décision est prise à partir de l'estimée fournis par le filtre et l'erreur est calculée. Finalement, les coefficients du filtre sont mis à jour à partir de cette erreur et du vecteur d'entrée X(n).

Sous forme mathématique, voici l'algorithme LMS [43]:

$$y(n) = W^H(n)X(n) \quad \textbf{(A-1)}$$

$$e(n) = d(n) - y(n) \quad \textbf{(A-2)}$$

$$W_k(n+1) = W_k(n) + \mu e^*(n)X_k(n) \quad \textbf{(A-3)}$$

Dans les équations précédentes, le H en exposant signifie la transposée hermitienne de la matrice (transposée et conjuguée de la matrice). Le symbole * en exposant désigne la conjuguée et le petit k en indice signifie le $k^{ième}$ élément de la matrice. La sortie du filtre

complexe est y(n), la sortie du module de décision d(n) et μ est le pas d'adaptation (fixe au cours de l'algorithme). Ce dernier détermine la convergence de l'algorithme.

En ce qui concerne l'algorithme RLS, celui-ci a une meilleure performance en terme de convergence. Cependant, sa complexité est plus grande que celle du LMS. La façon de procéder pour ce type d'égalisation est le même que celui LMS, c'est-à-dire un filtrage suivi d'une décision, d'un calcul d'erreur et d'une mise à jour des coefficients. Par contre, sa mise à jour des coefficients est relativement plus complexe. Cette complexité est visible dans la formulation mathématique de l'algorithme.

Sous forme mathématique, voici l'algorithme RLS [43]:

$$y(n) = W^H(n)X(n) \quad \textbf{(A-4)}$$

$$e(n) = d(n) - y(n) \quad \textbf{(A-5)}$$

$$\phi(n) = X^H(n)P(n) \quad \textbf{(A-6)}$$

$$K(n) = \frac{\phi^H(n)}{\lambda + \phi(n)X(n)} \quad \textbf{(A-7)}$$

$$W_k(n+1) = W_k(n) + K(n)e^*(n) \quad \textbf{(A-8)}$$

$$P(n+1) = \lambda^{-1}\left[P(n-1) - K(n)\phi(n)\right] \quad \textbf{(A-9)}$$

Dans les équations précédentes, le symbole ϕ désigne une matrice rangée de longueur N qui est un résultat intermédiaire de l'algorithme. La matrice P est l'inverse de l'autocorrelation du signal reçu et de dimensions NxN. La matrice K est un vecteur colonne de longueur N désigné sous le nom de vecteur de Kalman. Pour sa part, le symbole λ est le facteur d'oubli (scalaire) et inférieur ou égal à 1.

A.1 Égaliseur LTE

Il existe deux architectures principales d'égalisateur, l'architecture LTE (« Linear Transversal Equalizer ») et DFE (« Decision Feedback Equalizer »). Les deux figures suivantes illustrent ces architectures. La première architecture est la plus simple. Elle consiste en un filtre FIR, un module décision et d'un autre module pour la mise à jour des coefficients. Les modules de mise à jour des coefficients, dans ce cas comme dans celui de l'architecture DFE, peuvent être réalisés à l'aide d'un algorithme LMS ou RLS. Le symbole égalisé dans une architecture LTE est celui central du FIR, c'est-à-dire que l'interférence inter-symbole est supprimée pour des symboles précurseurs à celui égalisé et pour des symboles postcurseurs. Dans le cadre de cette architecture de même que pour celui du DFE, l'égaliseur doit être à priori entraîner de façon répétitive afin de faire converger celui-ci. L'entraînement consiste à transmettre une séquence de donnée connue et d'ajuster les coefficients en fonction de cette séquence. Pour réaliser un entraînement, il suffit de substituer la séquence de décision par la séquence d'entraînement dans l'égalisation.

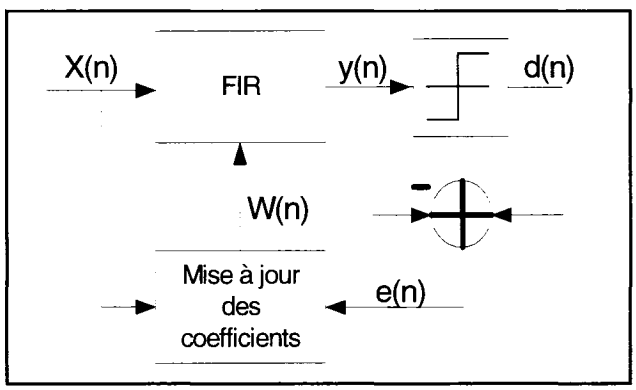

Figure A.1 : Architecture d'un égaliseur LTE-LMS

A.2 Égaliseur DFE

En ce qui concerne l'architecture DFE, celle-ci est constituée de deux filtres, soit un filtre avant (FFE, « FeedFoward Equalizer ») et un filtre arrière (FBE, « FeedBack Equalizer »). Ces deux filtres sont de type FIR. Le filtre avant sert à annuler les interférences précurseurs au symbole, tandis que le filtre arrière sert à annuler les interférences postcurseurs. Le filtre arrière est utilisé pour faire une contre-réaction avec la décision afin d'effectuer un estimé des interférences sachant les symboles transmis antérieurement. Contrairement à l'égaliseur linéaire transverse, le symbole égalisé dans cette architecture est celui à la fin du filtre avant et non, celui central. Il est à mentionner que la mise à jour des coefficients du filtre arrière se fait comme celle du filtre avant et selon le même algorithme, soit RLS ou LMS. La contre-réaction constituée du filtre arrière donne à ce type d'architecture, la forme d'un IIR. La forme globale de cet égaliseur étant de ce type, il se peut que celui-ci devienne instable et que celui-ci diverge. Afin de réduire cette instabilité, la longueur du filtre arrière est généralement courte, ce qui a pour effet de diminuer le nombre de pôle de l'égaliseur. Les performances de ce type d'architecture sont habituellement meilleures que celles de l'égaliseur LTE.

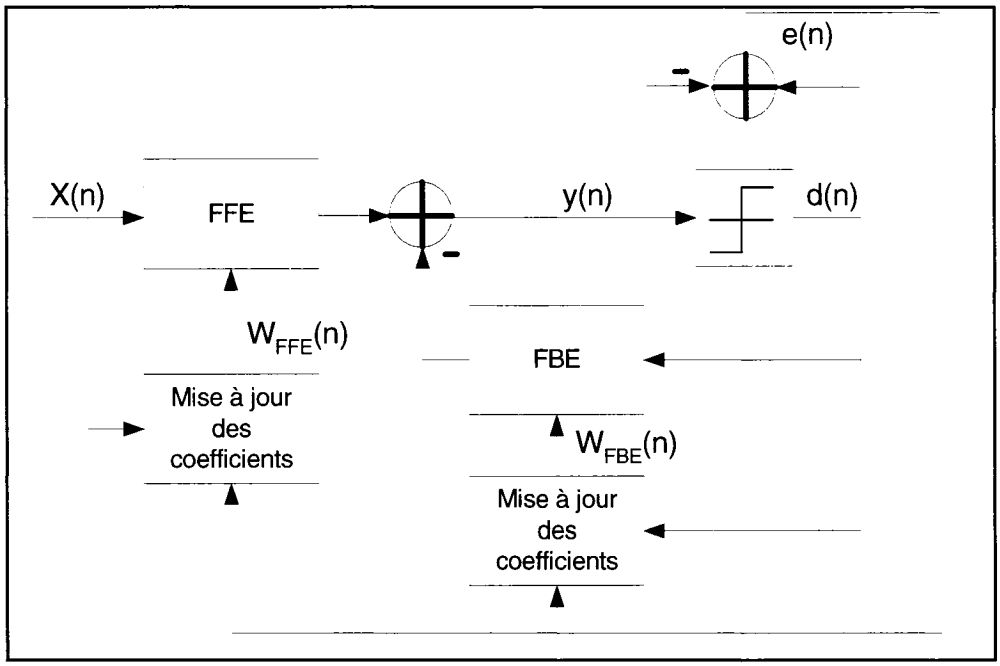

Figure A.2 : Architecture d'un égaliseur DFE-LMS

Il existe un autre type d'architecture en plus de ceux énumérés précédemment. Cet égaliseur diffère des autres, car son filtrage est effectué dans le domaine fréquentiel plutôt que temporel [13]. L'algorithme de mise à jour des coefficients le plus fréquemment utilisé pour ce type d'égaliseur est l'algorithme LMS. Pour des canaux de communications avec de long écho, ce type d'égaliseur est très performant. Celui-ci nécessite 3 FFT et 2 IFFT pour sa réalisation. Pour de plus amples renseignements sur ce type d'égaliseur, il faut consulter l'article [18].

L'égalisation est l'une des tâches les plus exigeante arithmétiquement pour les communications à haut débit [14]. Cette complexité peut se mesurer par le nombre de multiplication nécessaire pour cette tâche. Le cas le plus simple d'égaliseur est celui d'un égaliseur LTE-LMS. Pour le filtrage de ce type d'égaliseur, il faut N multiplications complexes, où N est le nombre de coefficients du filtre. Étant donnée que chaque multiplication complexe est l'équivalent de 4 multiplications réelles, il faut par conséquent 4N multiplications pour le filtrage pour chaque symbole filtré. Dans le

module de mise à jour des coefficients, il faut à priori multiplier l'erreur par le pas d'adaptation (réel). Il faut pour cette première étape 2 multiplications réelles. À ces deux multiplications, il faut rajouter N multiplications complexes pour le produit des données par l'erreur (4N multiplications réelles). En somme, pour réaliser l'égaliseur le plus simple, il faut un total de 8N+2 multiplications réelles. Maintenant, si un bilan des autres parties est effectué, il est possible de constater que le NCO (ou DDS) et les CIC ne nécessite aucune multiplication [1][16]. Que pour leur part, le mixer en nécessite 2, les filtres FIR en nécessite 2N (2 filtres de N coefficients), que la synchronisation de la porteuse en nécessite 3 pour le détecteur de phase et environ 2 pour le filtre de boucle. À la suite de ce bilan du nombre de multiplication, il est évident que la tâche la plus exigeante mathématiquement est l'égalisation.

ANNEXE B MATÉRIEL POUR LES UNITÉS SPÉCIALISÉES

Cette annexe donne un aperçu du code pour générer une unité spécialisée avec les deux technologies. Ainsi, cette annexe est constituée de deux parties, soit l'une concernant le processeur Xtensa et une autre concernant le processeur Nios. Le code fourni dans cette annexe décrit la fonctionnalité d'une unité spécialisée pour la réalisation d'un égaliseur LTE-LMS qui comprend un seul MAC complexe (MC). Le code pour la réalisation d'unité spécialisée pour les autres égaliseurs n'apparaît pas, car ce code est très similaire à celui présenté. Ainsi, la section B.1 dévoile le code TIE pour générer une unité spécialisée avec le processeur Xtensa et la section B.2 dévoile code VHDL pour l'unité spécialisée du processeur Nios.

B.1 Code TIE pour le processeur Xtensa (LTE-LMS avec 1 MC)

```
// Définition des opcodes des instructions spécialisées
opcode FIR   op2=4'b0000 CUST0
opcode SBPSK op2=4'b0001 CUST0
opcode TRAIN op2=4'b0010 CUST0
opcode ADAPT op2=4'b0011 CUST0
opcode UPDW  op2=4'b0100 CUST0
opcode DTRAIN op2=4'b0101 CUST0

// FIR data input
state FIR_D1 32
state FIR_D2 32
state FIR_D3 32
state FIR_D4 32
state FIR_D5 32
state FIR_D6 32
state FIR_D7 32
state FIR_D8 32

// Fir coefficients
state FIR_COEF0_RE 16
state FIR_COEF1_RE 16
state FIR_COEF2_RE 16
state FIR_COEF3_RE 16
state FIR_COEF4_RE 16
```

```
state FIR_COEF5_RE 16
state FIR_COEF6_RE 16
state FIR_COEF7_RE 16

state FIR_COEF0_IM 16
state FIR_COEF1_IM 16
state FIR_COEF2_IM 16
state FIR_COEF3_IM 16
state FIR_COEF4_IM 16
state FIR_COEF5_IM 16
state FIR_COEF6_IM 16
state FIR_COEF7_IM 16

// Données pour l'entraînement
state D_TRAIN0   16
state D_TRAIN1   16
state D_TRAIN2   16
state D_TRAIN3   16

// Pas d'adaptation
state EQ_MU    16

// Erreur quadratique
state EQ_ERROR_RE  16
state EQ_ERROR_IM  16

// Registre de sortie des multiplieurs
state MULT1_RES 32
state MULT2_RES 32
state MULT3_RES 32
state MULT4_RES 32

state FIR_RES_RE 32
state FIR_RES_IM 32

// Accès aux registres spécialisées
user_register FIRC0 0 {FIR_COEF0_RE[15:0], FIR_COEF0_IM[15:0]}
user_register FIRC1 1 {FIR_COEF1_RE[15:0], FIR_COEF1_IM[15:0]}
user_register FIRC2 2 {FIR_COEF2_RE[15:0], FIR_COEF2_IM[15:0]}
user_register FIRC3 3 {FIR_COEF3_RE[15:0], FIR_COEF3_IM[15:0]}
user_register FIRC4 4 {FIR_COEF4_RE[15:0], FIR_COEF4_IM[15:0]}
user_register FIRC5 5 {FIR_COEF5_RE[15:0], FIR_COEF5_IM[15:0]}
user_register FIRC6 6 {FIR_COEF6_RE[15:0], FIR_COEF6_IM[15:0]}
user_register FIRC7 7 {FIR_COEF7_RE[15:0], FIR_COEF7_IM[15:0]}

user_register EQERR 8 {EQ_ERROR_RE[15:0] , EQ_ERROR_IM[15:0]}
user_register EQMU  9 EQ_MU[15:0]
user_register DTRAIN3 10 D_TRAIN3[15:0]
user_register FIRD1 11 FIR_D1[31:0]
user_register FIRD2 12 FIR_D2[31:0]
user_register FIRD3 13 FIR_D3[31:0]
user_register FIRD4 14 FIR_D4[31:0]
user_register FIRD5 15 FIR_D5[31:0]
user_register FIRD6 16 FIR_D6[31:0]
user_register FIRD7 17 FIR_D7[31:0]
user_register FIRD8 18 FIR_D8[31:0]
```

```
// Decision prise par l'égaliseur
table decision 16 2 {
        4096, -4096
}

// Instruction pour le filtrage et la mise à jour
iclass eq_class {FIR, UPDW} {out arr, in ars, in art}
    {inout FIR_COEF0_RE, inout FIR_COEF1_RE, inout FIR_COEF2_RE, inout FIR_COEF3_RE,
     inout FIR_COEF0_IM, inout FIR_COEF1_IM, inout FIR_COEF2_IM, inout FIR_COEF3_IM,
     inout FIR_COEF4_RE, inout FIR_COEF5_RE, inout FIR_COEF6_RE, inout FIR_COEF7_RE,
     inout FIR_COEF4_IM, inout FIR_COEF5_IM, inout FIR_COEF6_IM, inout FIR_COEF7_IM,
     inout FIR_D1, inout FIR_D2, inout FIR_D3, inout FIR_D4, inout FIR_D5,
     inout FIR_D6, inout FIR_D7, inout FIR_D8, inout FIR_RES_RE, inout FIR_RES_IM,
     inout MULT1_RES, inout MULT2_RES, inout MULT3_RES,  inout MULT4_RES   }

semantic eq_class_sem {FIR, UPDW}{
    wire[31:0] tmp_re, tmp_im;
    wire[31:0] upd_re, upd_im;
    wire[15:0] mult1_in1, mult1_in2, mult2_in1, mult2_in2;
    wire[15:0] mult3_in1, mult3_in2, mult4_in1, mult4_in2;

    assign mult1_in1 = FIR ? TIEmux(art[2:0], FIR_COEF0_RE, FIR_COEF1_RE, FIR_COEF2_RE,
                                    FIR_COEF3_RE, FIR_COEF4_RE, FIR_COEF5_RE, FIR_COEF6_RE,
                                    FIR_COEF7_RE ) :
                                    TIEmux(art[2:0], FIR_D1[31:16], FIR_D2[31:16], FIR_D3[31:16],
                                    FIR_D4[31:16], FIR_D5[31:16], FIR_D6[31:16], FIR_D7[31:16],
                                    FIR_D8[31:16]);
    assign mult1_in2 = FIR ? TIEmux(art[2:0], ars[31:16]   , FIR_D1[31:16], FIR_D2[31:16],
                                    FIR_D3[31:16], FIR_D4[31:16], FIR_D5[31:16], FIR_D6[31:16],
                                    FIR_D7[31:16]) :
                                    ars[31:16];

    assign mult2_in1 = FIR ? TIEmux(art[2:0], FIR_COEF0_RE, FIR_COEF1_RE, FIR_COEF2_RE,
                                    FIR_COEF3_RE, FIR_COEF4_RE, FIR_COEF5_RE, FIR_COEF6_RE,
                                    FIR_COEF7_RE ):
                                    TIEmux(art[2:0], FIR_D1[31:16], FIR_D2[31:16], FIR_D3[31:16],
                                    FIR_D4[31:16],  FIR_D5[31:16], FIR_D6[31:16], FIR_D7[31:16],
                                    FIR_D8[31:16]);
    assign mult2_in2 = FIR ? TIEmux(art[2:0], ars[15:0 ]  , FIR_D1[15:0], FIR_D2[15:0], FIR_D3[15:0],
                                    FIR_D4[15:0], FIR_D5[15:0], FIR_D6[15:0], FIR_D7[15:0]) :
                                    ars[15:0 ];

    assign mult3_in1 = FIR ? TIEmux(art[2:0], FIR_COEF0_IM, FIR_COEF1_IM, FIR_COEF2_IM,
                                    FIR_COEF3_IM,  FIR_COEF4_IM, FIR_COEF5_IM, FIR_COEF6_IM,
                                    FIR_COEF7_IM) :
                                    TIEmux(art[2:0], FIR_D1[15:0 ], FIR_D2[15:0 ], FIR_D3[15:0 ],
                                    FIR_D4[15:0 ], FIR_D5[15:0 ], FIR_D6[15:0 ], FIR_D7[15:0 ],
                                    FIR_D8[15:0 ]);
    assign mult3_in2 = FIR ? TIEmux(art[2:0], ars[31:16]   , FIR_D1[31:16], FIR_D2[31:16],
                                    FIR_D3[31:16], FIR_D4[31:16], FIR_D5[31:16], FIR_D6[31:16],
                                    FIR_D7[31:16]) :
                                    ars[31:16];

    assign mult4_in1 = FIR ? TIEmux(art[2:0], FIR_COEF0_IM, FIR_COEF1_IM, FIR_COEF2_IM,
                                    FIR_COEF3_IM, FIR_COEF4_IM, FIR_COEF5_IM, FIR_COEF6_IM,
                                    FIR_COEF7_IM) :
                                    TIEmux(art[2:0], FIR_D1[15:0 ], FIR_D2[15:0 ], FIR_D3[15:0 ],
```

```
                                FIR_D4[15:0 ], FIR_D5[15:0 ], FIR_D6[15:0 ], FIR_D7[15:0 ],
                                FIR_D8[15:0 ]);
    assign mult4_in2 = FIR ? TIEmux(art[2:0], ars[15:0 ] , FIR_D1[15:0], FIR_D2[15:0], FIR_D3[15:0],
                                FIR_D4[15:0], FIR_D5[15:0], FIR_D6[15:0], FIR_D7[15:0]):
                                ars[15:0 ];



    assign MULT1_RES  = TIEmul(mult1_in1, mult1_in2, 1'b1); // RE*RE
    assign MULT2_RES  = TIEmul(mult2_in1, mult2_in2, 1'b1); // RE*IM
    assign MULT3_RES  = TIEmul(mult3_in1, mult3_in2, 1'b1); // IM*RE
    assign MULT4_RES  = TIEmul(mult4_in1, mult4_in2, 1'b1); // IM*IM

    assign FIR_D8 = (FIR & art[3:0] == 8 )? FIR_D7 : FIR_D8;
    assign FIR_D7 = (FIR & art[3:0] == 8 )? FIR_D6 : FIR_D7;
    assign FIR_D6 = (FIR & art[3:0] == 8 )? FIR_D5 : FIR_D6;
    assign FIR_D5 = (FIR & art[3:0] == 8 )? FIR_D4 : FIR_D5;
    assign FIR_D4 = (FIR & art[3:0] == 8 )? FIR_D3 : FIR_D4;
    assign FIR_D3 = (FIR & art[3:0] == 8 )? FIR_D2 : FIR_D3;
    assign FIR_D2 = (FIR & art[3:0] == 8 )? FIR_D1 : FIR_D2;
    assign FIR_D1 = (FIR & art[3:0] == 8 )? ars    : FIR_D1;

    assign tmp_re = (MULT1_RES + MULT4_RES) + (art[3:0] == 1 ? 32'b0 : FIR_RES_RE );
    assign tmp_im = (MULT2_RES - MULT3_RES) + (art[3:0] == 1 ? 32'b0 : FIR_RES_IM );

    assign FIR_RES_RE = tmp_re;
    assign FIR_RES_IM = tmp_im;

    assign upd_re = MULT1_RES + MULT4_RES;
    assign upd_im = MULT3_RES - MULT2_RES;

    assign FIR_COEF0_RE = (UPDW & art[3:0] == 1) ? FIR_COEF0_RE + upd_re[31:16] : FIR_COEF0_RE ;
    assign FIR_COEF0_IM = (UPDW & art[3:0] == 1) ? FIR_COEF0_IM + upd_im[31:16] : FIR_COEF0_IM ;
    assign FIR_COEF1_RE = (UPDW & art[3:0] == 2) ? FIR_COEF1_RE + upd_re[31:16] : FIR_COEF1_RE ;
    assign FIR_COEF1_IM = (UPDW & art[3:0] == 2) ? FIR_COEF1_IM + upd_im[31:16] : FIR_COEF1_IM ;
    assign FIR_COEF2_RE = (UPDW & art[3:0] == 3) ? FIR_COEF2_RE + upd_re[31:16] : FIR_COEF2_RE ;
    assign FIR_COEF2_IM = (UPDW & art[3:0] == 3) ? FIR_COEF2_IM + upd_im[31:16] : FIR_COEF2_IM ;
    assign FIR_COEF3_RE = (UPDW & art[3:0] == 4) ? FIR_COEF3_RE + upd_re[31:16] : FIR_COEF3_RE ;
    assign FIR_COEF3_IM = (UPDW & art[3:0] == 4) ? FIR_COEF3_IM + upd_im[31:16] : FIR_COEF3_IM ;

    assign FIR_COEF4_RE = (UPDW & art[3:0] == 5) ? FIR_COEF4_RE + upd_re[31:16] : FIR_COEF4_RE ;
    assign FIR_COEF4_IM = (UPDW & art[3:0] == 5) ? FIR_COEF4_IM + upd_im[31:16] : FIR_COEF4_IM ;
    assign FIR_COEF5_RE = (UPDW & art[3:0] == 6) ? FIR_COEF5_RE + upd_re[31:16] : FIR_COEF5_RE ;
    assign FIR_COEF5_IM = (UPDW & art[3:0] == 6) ? FIR_COEF5_IM + upd_im[31:16] : FIR_COEF5_IM ;
    assign FIR_COEF6_RE = (UPDW & art[3:0] == 7) ? FIR_COEF6_RE + upd_re[31:16] : FIR_COEF6_RE ;
    assign FIR_COEF6_IM = (UPDW & art[3:0] == 7) ? FIR_COEF6_IM + upd_im[31:16] : FIR_COEF6_IM ;
    assign FIR_COEF7_RE = (UPDW & art[3:0] == 8) ? FIR_COEF7_RE + upd_re[31:16] : FIR_COEF7_RE ;
    assign FIR_COEF7_IM = (UPDW & art[3:0] == 8) ? FIR_COEF7_IM + upd_im[31:16] : FIR_COEF7_IM ;

    assign arr = {tmp_re[31:16], tmp_im[31:16]};
}

schedule eq_class_sem {FIR, UPDW}{

    use FIR_COEF0_RE 1;
    use FIR_COEF1_RE 1;
```

```
use FIR_COEF2_RE 1;
use FIR_COEF3_RE 1;
use FIR_COEF4_RE 1;
use FIR_COEF5_RE 1;
use FIR_COEF6_RE 1;
use FIR_COEF7_RE 1;
use FIR_COEF0_IM 1;
use FIR_COEF1_IM 1;
use FIR_COEF2_IM 1;
use FIR_COEF3_IM 1;
use FIR_COEF4_IM 1;
use FIR_COEF5_IM 1;
use FIR_COEF6_IM 1;
use FIR_COEF7_IM 1;

use FIR_D8 1;
use FIR_D7 1;
use FIR_D6 1;
use FIR_D5 1;
use FIR_D4 1;
use FIR_D3 1;
use FIR_D2 1;
use FIR_D1 1;

use MULT1_RES 1;
use MULT2_RES 1;
use MULT3_RES 1;
use MULT4_RES 1;

def FIR_D8 2;
def FIR_D7 2;
def FIR_D6 2;
def FIR_D5 2;
def FIR_D4 2;
def FIR_D3 2;
def FIR_D2 2;
def FIR_D1 2;

def MULT1_RES 2;
def MULT2_RES 2;
def MULT3_RES 2;
def MULT4_RES 2;

def FIR_COEF0_RE 2;
def FIR_COEF1_RE 2;
def FIR_COEF2_RE 2;
def FIR_COEF3_RE 2;
def FIR_COEF4_RE 2;
def FIR_COEF5_RE 2;
def FIR_COEF6_RE 2;
def FIR_COEF7_RE 2;
def FIR_COEF0_IM 2;
def FIR_COEF1_IM 2;
def FIR_COEF2_IM 2;
def FIR_COEF3_IM 2;
def FIR_COEF4_IM 2;
def FIR_COEF5_IM 2;
```

```
    def FIR_COEF6_IM 2;
    def FIR_COEF7_IM 2;

    def FIR_RES_RE 2;
    def FIR_RES_IM 2;
    def arr 2;

}

// Instruction pour la décision sur le symbole
iclass slic_class {SBPSK, TRAIN} {out arr, in ars}
    {out EQ_ERROR_RE, out EQ_ERROR_IM, in D_TRAIN3 }

semantic slic_class_sem {SBPSK, TRAIN}{

    wire[31:0] tmp1;

    assign tmp1 = SBPSK ? (ars[31] ? {decision[1] ,16'b0} : {decision[0] ,16'b0}) : {D_TRAIN3 ,16'b0};

    assign EQ_ERROR_RE = tmp1[31:16] - ars[31:16];
    assign EQ_ERROR_IM = tmp1[15:0 ] - ars[15:0 ];

    assign arr = tmp1;

}

// Décalage des données d'entraînement
iclass dtrain_class {DTRAIN} { in ars}
    {inout D_TRAIN0, inout D_TRAIN1, inout D_TRAIN2, out D_TRAIN3 }

semantic dtrain_class {DTRAIN}{

    assign D_TRAIN3 = D_TRAIN2;
    assign D_TRAIN2 = D_TRAIN1;
    assign D_TRAIN1 = D_TRAIN0;
    assign D_TRAIN0 = ars[15:0];

}

// Instruction pour le calcul le produit mu*erreur
iclass adapt_class {ADAPT} {out arr}
    {in EQ_MU, in EQ_ERROR_RE, in EQ_ERROR_IM }

semantic adapt_class {ADAPT}{

    wire[31:0] tmp1, tmp2;
    wire[31:0] mu_error;

    assign tmp1 = TIEmul(EQ_MU, EQ_ERROR_RE, 1'b1);
    assign tmp2 = TIEmul(EQ_MU, EQ_ERROR_IM, 1'b1);
    assign mu_error = {tmp1[31:16], tmp2[31:16]};

    assign arr = mu_error;

}

schedule adapt_class {ADAPT}{
```

```
    use EQ_MU 1;
    use EQ_ERROR_RE 1;
    use EQ_ERROR_IM 1;

    def arr 2;
}
```

B.2 Code VHDL pour le NIOS (LTE-LMS avec 1 MC)

```
--------------------------------------------------------------------------------
--
--      Fichier : LTE_1mc.vhd
--
--      Description : Coprocesseur contenant 1 multiplieur complexe utilisé pour
--                    accélérer un égaliseur LTE-LMS (8 taps) avec le processeur NIOS.
--
--      Auteur  : Bruno Tanguay
--
--      Date    : 10 juin 2004
--
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_signed.all;
USE IEEE.std_logic_1164.all;


ENTITY LTE_1mc IS

        PORT(
                clk     : IN STD_LOGIC; -- CPU's master-input clk <required for multi-cycle>
                reset   : IN STD_LOGIC; -- CPU's master asynchronous reset <req. for multi-cycle>
                clk_en  : IN STD_LOGIC; -- Clock-qualifier <required for multi-cycle>
                start   : IN STD_LOGIC; -- True when this instr. issues <required for multi-cycle>
                dataa   : IN STD_LOGIC_VECTOR (31 DOWNTO 0); -- operand A <always required>
                datab   : IN STD_LOGIC_VECTOR (31 DOWNTO 0); -- operand B <optional>
                prefix  : IN STD_LOGIC_VECTOR (10 DOWNTO 0); -- prefix <optional>
                result  : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
        );

END LTE_1mc;

ARCHITECTURE LTE_1mc_rtl OF LTE_1mc IS

        -- decisions sur le symbole BPSK
        CONSTANT BPSK_P : std_logic_vector( 15 downto 0):= X"1000";  -- dec =  1
        CONSTANT BPSK_N : std_logic_vector( 15 downto 0):= X"F000";  -- dec = -1
```

```
        -- type de signaux pour les coefficients
        type coeff_real is array(7 downto 0) of std_logic_vector( 15 downto 0);
        type coeff_imag is array(7 downto 0) of std_logic_vector( 15 downto 0);

        -- type de signaux pour les données
        type data_real  is array(7 downto 0) of std_logic_vector( 15 downto 0);
        type data_imag  is array(7 downto 0) of std_logic_vector( 15 downto 0);

        -- type de signaux pour données d'entraînement
        type data_train is array(3 downto 0) of std_logic_vector( 15 downto 0);

        -- registres des coefficients
        signal FIR_COEF_RE : coeff_real;
        signal FIR_COEF_IM : coeff_imag;

        -- registres des de données
        signal FIR_D_RE    : data_real;
        signal FIR_D_IM    : data_imag;

        -- registres pour entraînement
        signal DTRAIN      : data_train;

        -- registre pour MU et Error
        signal EQ_MU       : std_logic_vector(15 downto 0);
        signal EQ_ERROR_RE : std_logic_vector(15 downto 0);
        signal EQ_ERROR_IM : std_logic_vector(15 downto 0);

        -- registres d'entrée des multiplieurs
        signal mult1_in1   : std_logic_vector(15 downto 0);
        signal mult1_in2   : std_logic_vector(15 downto 0);
        signal mult2_in1   : std_logic_vector(15 downto 0);
        signal mult2_in2   : std_logic_vector(15 downto 0);
        signal mult3_in1   : std_logic_vector(15 downto 0);
        signal mult3_in2   : std_logic_vector(15 downto 0);
        signal mult4_in1   : std_logic_vector(15 downto 0);
        signal mult4_in2   : std_logic_vector(15 downto 0);

        -- registres de sorties pour les multiplications
        signal MULT1_RES   : std_logic_vector(31 downto 0);
        signal MULT2_RES   : std_logic_vector(31 downto 0);
        signal MULT3_RES   : std_logic_vector(31 downto 0);
        signal MULT4_RES   : std_logic_vector(31 downto 0);

        -- resultats temporaires pour le filtrage
        signal FIR_RES_RE  : std_logic_vector(31 downto 0);
        signal FIR_RES_IM  : std_logic_vector(31 downto 0);

BEGIN   -- architecture LTE_1mc_rtl

        do_equalize : process (clk, reset)

        variable decision  : std_logic_vector(15 downto 0);
        variable tmp_real  : std_logic_vector(31 downto 0);
        variable tmp_imag  : std_logic_vector(31 downto 0);

        begin  -- PROCESS do_equalize
```

```
if( reset = '1') then

-- remise à zéro de tous les registres
for index in 0 to 7 loop
    FIR_COEF_RE(index) <= (others => '0');
    FIR_COEF_IM(index) <= (others => '0');
    FIR_D_RE(index)    <= (others => '0');
    FIR_D_IM(index)    <= (others => '0');
end loop;

for index in 0 to 3 loop
    DTRAIN(index)      <= (others => '0');
end loop;

mult1_in1  <= (others => '0');
mult1_in2  <= (others => '0');
mult2_in1  <= (others => '0');
mult2_in2  <= (others => '0');
mult3_in1  <= (others => '0');
mult3_in2  <= (others => '0');
mult4_in1  <= (others => '0');
mult4_in2  <= (others => '0');

MULT1_RES  <= (others => '0');
MULT2_RES  <= (others => '0');
MULT3_RES  <= (others => '0');
MULT4_RES  <= (others => '0');

FIR_RES_RE <= (others => '0');
FIR_RES_IM <= (others => '0');

EQ_MU      <= (others => '0');
EQ_ERROR_RE <= (others => '0');
EQ_ERROR_IM <= (others => '0');

elsif(clk'event and clk = '1') then

  if(start = '1')then

     case prefix(2 downto 0) is

     -- FIR Instructions
     -- Filtrage des symboles
     when "000" =>
         -- Sélection des entrées des multiplieurs
         case datab(3 downto 0) is

          when X"0" =>
                mult1_in1 <= FIR_COEF_RE(0);
                mult1_in2 <= dataa(31 downto 16);
                mult2_in1 <= FIR_COEF_RE(0);
                mult2_in2 <= dataa(15 downto 0 );
                mult3_in1 <= FIR_COEF_IM(0);
                mult3_in2 <= dataa(31 downto 16);
                mult4_in1 <= FIR_COEF_IM(0);
                mult4_in2 <= dataa(15 downto 0 );
```

```
        tmp_real := (others => '0');
        tmp_imag := (others => '0');

    when X"1" =>
        mult1_in1 <= FIR_COEF_RE(1);
        mult1_in2 <= FIR_D_RE(0);
        mult2_in1 <= FIR_COEF_RE(1);
        mult2_in2 <= FIR_D_IM(0);
        mult3_in1 <= FIR_COEF_IM(1);
        mult3_in2 <= FIR_D_RE(0);
        mult4_in1 <= FIR_COEF_IM(1);
        mult4_in2 <= FIR_D_IM(0);

        tmp_real := (others => '0');
        tmp_imag := (others => '0');

    when X"2" =>
        mult1_in1 <= FIR_COEF_RE(2);
        mult1_in2 <= FIR_D_RE(1);
        mult2_in1 <= FIR_COEF_RE(2);
        mult2_in2 <= FIR_D_IM(1);
        mult3_in1 <= FIR_COEF_IM(2);
        mult3_in2 <= FIR_D_RE(1);
        mult4_in1 <= FIR_COEF_IM(2);
        mult4_in2 <= FIR_D_IM(1);

        tmp_real := (others => '0');
        tmp_imag := (others => '0');

    when X"3" =>
        mult1_in1 <= FIR_COEF_RE(3);
        mult1_in2 <= FIR_D_RE(2);
        mult2_in1 <= FIR_COEF_RE(3);
        mult2_in2 <= FIR_D_IM(2);
        mult3_in1 <= FIR_COEF_IM(3);
        mult3_in2 <= FIR_D_RE(2);
        mult4_in1 <= FIR_COEF_IM(3);
        mult4_in2 <= FIR_D_IM(2);

        tmp_real := FIR_RES_RE;
        tmp_imag := FIR_RES_IM;

    when X"4" =>
        mult1_in1 <= FIR_COEF_RE(4);
        mult1_in2 <= FIR_D_RE(3);
        mult2_in1 <= FIR_COEF_RE(4);
        mult2_in2 <= FIR_D_IM(3);
        mult3_in1 <= FIR_COEF_IM(4);
        mult3_in2 <= FIR_D_RE(3);
        mult4_in1 <= FIR_COEF_IM(4);
        mult4_in2 <= FIR_D_IM(3);

        tmp_real := FIR_RES_RE;
        tmp_imag := FIR_RES_IM;

    when X"5" =>
```

```
        mult1_in1 <= FIR_COEF_RE(5);
        mult1_in2 <= FIR_D_RE(4);
        mult2_in1 <= FIR_COEF_RE(5);
        mult2_in2 <= FIR_D_IM(4);
        mult3_in1 <= FIR_COEF_IM(5);
        mult3_in2 <= FIR_D_RE(4);
        mult4_in1 <= FIR_COEF_IM(5);
        mult4_in2 <= FIR_D_IM(4);

        tmp_real := FIR_RES_RE;
        tmp_imag := FIR_RES_IM;

    when X"6" =>
        mult1_in1 <= FIR_COEF_RE(6);
        mult1_in2 <= FIR_D_RE(5);
        mult2_in1 <= FIR_COEF_RE(6);
        mult2_in2 <= FIR_D_IM(5);
        mult3_in1 <= FIR_COEF_IM(6);
        mult3_in2 <= FIR_D_RE(5);
        mult4_in1 <= FIR_COEF_IM(6);
        mult4_in2 <= FIR_D_IM(5);

        tmp_real := FIR_RES_RE;
        tmp_imag := FIR_RES_IM;

    when X"7" =>
        mult1_in1 <= FIR_COEF_RE(7);
        mult1_in2 <= FIR_D_RE(6);
        mult2_in1 <= FIR_COEF_RE(7);
        mult2_in2 <= FIR_D_IM(6);
        mult3_in1 <= FIR_COEF_IM(7);
        mult3_in2 <= FIR_D_RE(6);
        mult4_in1 <= FIR_COEF_IM(7);
        mult4_in2 <= FIR_D_IM(6);

        tmp_real := FIR_RES_RE;
        tmp_imag := FIR_RES_IM;

        for index in 7 downto 1 loop
            FIR_D_RE(index) <= FIR_D_RE(index-1);
            FIR_D_IM(index) <= FIR_D_IM(index-1);
        end loop;

        FIR_D_RE(0) <= dataa(31 downto 16);
        FIR_D_IM(0) <= dataa(15 downto 0 );

    when X"8" =>
        mult1_in1 <= (others => '0');
        mult1_in2 <= (others => '0');
        mult2_in1 <= (others => '0');
        mult2_in2 <= (others => '0');
        mult3_in1 <= (others => '0');
        mult3_in2 <= (others => '0');
        mult4_in1 <= (others => '0');
        mult4_in2 <= (others => '0');

        tmp_real := FIR_RES_RE;
```

```
            tmp_imag := FIR_RES_IM;

        when others =>
            mult1_in1 <= (others => '0');
            mult1_in2 <= (others => '0');
            mult2_in1 <= (others => '0');
            mult2_in2 <= (others => '0');
            mult3_in1 <= (others => '0');
            mult3_in2 <= (others => '0');
            mult4_in1 <= (others => '0');
            mult4_in2 <= (others => '0');

            tmp_real := FIR_RES_RE;
            tmp_imag := FIR_RES_IM;

        end case;

        -- multiplications 16x16 des différents
        -- parties réelles et imaginaire
        MULT1_RES <= mult1_in1 * mult1_in2;
        MULT2_RES <= mult2_in1 * mult2_in2;
        MULT3_RES <= mult3_in1 * mult3_in2;
        MULT4_RES <= mult4_in1 * mult4_in2;

        -- addition de la partie réelle et imaginaire
        tmp_real := (MULT1_RES + MULT4_RES) + tmp_real;
        tmp_imag := (MULT2_RES - MULT3_RES) + tmp_imag;

        -- accumulateur de la partie réelle et imaginaire
        FIR_RES_RE <= tmp_real;
        FIR_RES_IM <= tmp_imag;

        result <= tmp_real(31 downto 16) & tmp_imag(31 downto 16);


    -- DTRAIN Instruction
    -- Décalage des données d'entraînement
    when "001" =>

        for index in 3 downto 1 loop
            DTRAIN(index) <= DTRAIN(index-1);
        end loop;
        DTRAIN(0) <= dataa(15 downto 0);

    -- SBPSK Instruction
    -- Prise de décision (slicer) sur le symbole BPSK
    -- et calcul de l'erreur EQ
    when "010" =>       -- SBPSK

        if(dataa(31) = '0') then
            decision := BPSK_P;
        else
            decision := BPSK_N;
        end if;

        -- Calcul de l'erreur
        EQ_ERROR_RE <= decision - dataa(31 downto 16);
```

```
        EQ_ERROR_IM <= - dataa(15 downto 0);

        result <= decision & X"0000";

-- TRAIN Instruction
-- Entraînement et calcul de l'erreur EQ
when "011" =>

        -- Calcul de l'erreur
        EQ_ERROR_RE <= DTRAIN(3) - dataa(31 downto 16);
        EQ_ERROR_IM <= - dataa(15 downto 0);

        result <= DTRAIN(3) & X"0000";

-- ADAPT Instruction
-- Calcul de la multiplication MU par erreur
when "100" =>

        tmp_real := EQ_MU * EQ_ERROR_RE;
        tmp_imag := EQ_MU * EQ_ERROR_IM;

        result <= tmp_real(31 downto 16) & tmp_imag(31 downto 16);

-- UPDW Instruction
-- Mise à jour des coefficients
when "101" =>   -- UPDW

        -- Sélection des entrées des multiplieurs
        case datab(3 downto 0) is
                when X"0" =>
                        mult1_in1 <= FIR_D_RE(0);
                        mult1_in2 <= dataa(31 downto 16);
                        mult2_in1 <= FIR_D_RE(0);
                        mult2_in2 <= dataa(15 downto 0 );
                        mult3_in1 <= FIR_D_IM(0);
                        mult3_in2 <= dataa(31 downto 16);
                        mult4_in1 <= FIR_D_IM(0);
                        mult4_in2 <= dataa(15 downto 0 );

                when X"1" =>
                        mult1_in1 <= FIR_D_RE(1);
                        mult1_in2 <= dataa(31 downto 16);
                        mult2_in1 <= FIR_D_RE(1);
                        mult2_in2 <= dataa(15 downto 0 );
                        mult3_in1 <= FIR_D_IM(1);
                        mult3_in2 <= dataa(31 downto 16);
                        mult4_in1 <= FIR_D_IM(1);
                        mult4_in2 <= dataa(15 downto 0 );

                when X"2" =>
                        mult1_in1 <= FIR_D_RE(2);
                        mult1_in2 <= dataa(31 downto 16);
                        mult2_in1 <= FIR_D_RE(2);
                        mult2_in2 <= dataa(15 downto 0 );
                        mult3_in1 <= FIR_D_IM(2);
                        mult3_in2 <= dataa(31 downto 16);
                        mult4_in1 <= FIR_D_IM(2);
```

```
        mult4_in2 <= dataa(15 downto 0 );

when X"3" =>
        mult1_in1 <= FIR_D_RE(3);
        mult1_in2 <= dataa(31 downto 16);
        mult2_in1 <= FIR_D_RE(3);
        mult2_in2 <= dataa(15 downto 0 );
        mult3_in1 <= FIR_D_IM(3);
        mult3_in2 <= dataa(31 downto 16);
        mult4_in1 <= FIR_D_IM(3);
        mult4_in2 <= dataa(15 downto 0 );

when X"4" =>
        mult1_in1 <= FIR_D_RE(4);
        mult1_in2 <= dataa(31 downto 16);
        mult2_in1 <= FIR_D_RE(4);
        mult2_in2 <= dataa(15 downto 0 );
        mult3_in1 <= FIR_D_IM(4);
        mult3_in2 <= dataa(31 downto 16);
        mult4_in1 <= FIR_D_IM(4);
        mult4_in2 <= dataa(15 downto 0 );

when X"5" =>
        mult1_in1 <= FIR_D_RE(5);
        mult1_in2 <= dataa(31 downto 16);
        mult2_in1 <= FIR_D_RE(5);
        mult2_in2 <= dataa(15 downto 0 );
        mult3_in1 <= FIR_D_IM(5);
        mult3_in2 <= dataa(31 downto 16);
        mult4_in1 <= FIR_D_IM(5);
        mult4_in2 <= dataa(15 downto 0 );

when X"6" =>
        mult1_in1 <= FIR_D_RE(6);
        mult1_in2 <= dataa(31 downto 16);
        mult2_in1 <= FIR_D_RE(6);
        mult2_in2 <= dataa(15 downto 0 );
        mult3_in1 <= FIR_D_IM(6);
        mult3_in2 <= dataa(31 downto 16);
        mult4_in1 <= FIR_D_IM(6);
        mult4_in2 <= dataa(15 downto 0 );

when X"7" =>
        mult1_in1 <= FIR_D_RE(7);
        mult1_in2 <= dataa(31 downto 16);
        mult2_in1 <= FIR_D_RE(7);
        mult2_in2 <= dataa(15 downto 0 );
        mult3_in1 <= FIR_D_IM(7);
        mult3_in2 <= dataa(31 downto 16);
        mult4_in1 <= FIR_D_IM(7);
        mult4_in2 <= dataa(15 downto 0 );

when others =>
        mult1_in1 <= (others=> '0');
        mult1_in2 <= (others=> '0');
        mult2_in1 <= (others=> '0');
        mult2_in2 <= (others=> '0');
```

```
                mult3_in1 <= (others=> '0');
                mult3_in2 <= (others=> '0');
                mult4_in1 <= (others=> '0');
                mult4_in2 <= (others=> '0');

        end case;

        -- multiplications 16x16 des différents
        -- parties réelles et imaginaire
        MULT1_RES <= mult1_in1 * mult1_in2;
        MULT2_RES <= mult2_in1 * mult2_in2;
        MULT3_RES <= mult3_in1 * mult3_in2;
        MULT4_RES <= mult4_in1 * mult4_in2;

        -- addition de la partie réelle et imaginaire
        tmp_real := (MULT1_RES + MULT4_RES);
        tmp_imag := (MULT3_RES - MULT2_RES);

        -- Mise à jour des coefficients
        case datab(3 downto 0) is
            when X"2" =>
                FIR_COEF_RE(0) <= FIR_COEF_RE(0) + tmp_real(31 downto 16);
                FIR_COEF_IM(0) <= FIR_COEF_IM(0) + tmp_imag(31 downto 16);

            when X"3" =>
                FIR_COEF_RE(1) <= FIR_COEF_RE(1) + tmp_real(31 downto 16);
                FIR_COEF_IM(1) <= FIR_COEF_IM(1) + tmp_imag(31 downto 16);

            when X"4" =>
                FIR_COEF_RE(2) <= FIR_COEF_RE(2) + tmp_real(31 downto 16);
                FIR_COEF_IM(2) <= FIR_COEF_IM(2) + tmp_imag(31 downto 16);

            when X"5" =>
                FIR_COEF_RE(3) <= FIR_COEF_RE(3) + tmp_real(31 downto 16);
                FIR_COEF_IM(3) <= FIR_COEF_IM(3) + tmp_imag(31 downto 16);

            when X"6" =>
                FIR_COEF_RE(4) <= FIR_COEF_RE(4) + tmp_real(31 downto 16);
                FIR_COEF_IM(4) <= FIR_COEF_IM(4) + tmp_imag(31 downto 16);

            when X"7" =>
                FIR_COEF_RE(5) <= FIR_COEF_RE(5) + tmp_real(31 downto 16);
                FIR_COEF_IM(5) <= FIR_COEF_IM(5) + tmp_imag(31 downto 16);

            when X"8" =>
                FIR_COEF_RE(6) <= FIR_COEF_RE(6) + tmp_real(31 downto 16);
                FIR_COEF_IM(6) <= FIR_COEF_IM(6) + tmp_imag(31 downto 16);

            when X"9" =>
                FIR_COEF_RE(7) <= FIR_COEF_RE(7) + tmp_real(31 downto 16);
                FIR_COEF_IM(7) <= FIR_COEF_IM(7) + tmp_imag(31 downto 16);

            when others => null;
        end case;

    -- READ Instruction
    -- Lecture des registres
```

```
                when "110" =>

                    if(datab = 0) then
                        result <=  EQ_MU & X"0000";
                    elsif(datab = 1) then
                        result <= EQ_ERROR_RE & EQ_ERROR_IM ;
                    elsif(datab >= 2 AND datab <=9 ) then              -- weigths
                        result <= FIR_COEF_RE(conv_integer(datab-2)) & FIR_COEF_IM(conv_integer(datab-2));
                    elsif(datab >= 10 AND datab <= 17 ) then    -- data
                        result <= FIR_D_RE(conv_integer(datab-10)) & FIR_D_IM(conv_integer(datab-10));
                    elsif(datab >= 18 AND datab <= 21 )then
                        result <= DTRAIN(conv_integer(datab-18)) & X"0000";
                    end if;

                -- WRITE Instruction
                -- Écriture des registres
                when "111" =>

                    if(datab = 0) then
                        EQ_MU <= dataa(31 downto 16);
                    elsif(datab = 1) then
                        EQ_ERROR_RE <= dataa(31 downto 16);
                        EQ_ERROR_IM <= dataa(15 downto 0);
                    elsif(datab >= 2 AND datab <=9 ) then              -- weigths
                        FIR_COEF_RE(conv_integer(datab-2)) <= dataa(31 downto 16);
                        FIR_COEF_IM(conv_integer(datab-2)) <= dataa(15 downto 0);
                    elsif(datab >= 10 AND datab <= 17 ) then    -- data complex
                        FIR_D_RE(conv_integer(datab-10)) <= dataa(31 downto 16);
                        FIR_D_IM(conv_integer(datab-10)) <= dataa(15 downto 0);
                    elsif(datab >= 18 AND datab <= 21 )then
                        DTRAIN(conv_integer(datab-18)) <= dataa(31 downto 16);
                    end if;

                end case;               -- prefix
            end if;                     -- start
        end if;             -- reset

    end process do_equalize;

END LTE_1mc_rtl;
```

ANNEXE C LOGICIEL POUR LES ÉGALISEURS

Cette annexe dévoile le code logiciel pour la réalisation d'égaliseur LTE-LMS, sans ou avec une unité spécialisée. Il est à mentionné que les égaliseurs LTE-LMS réalisé dans le cadre de ces travaux contient 8 coefficients. Le code logiciel pour la réalisation d'égaliseur DFE-LMS n'est pas présenté dans cette section, car il ressemble fortement à celui d'un égaliseur LTE-LMS. Cette annexe est constituée principalement de trois sections. La première section (C.1) exhibe les fichiers originaux pour la réalisation d'un égaliseur LTE-LMS avec un processeur sans unité spécialisée, ce code peut être exécuté avec les deux technologies. La seconde section (C.2) démontre les fichiers pour un processeur Xtensa avec une unité spécialisée qui comprend un seul MAC complexe (MC). Pour sa part, la dernière section démontre les fichiers pour un processeur Nios avec une unité spécialisée qui comprend un seul MAC complexe (MC).

C.1 Programme en C original

Cette section comprend les fichiers suivants : main.cc, equalizer_LMS.h, equalizer_LMS.c, equalizer.h et equalizer.c. Les fichiers sont présentés dans un ordre hiérarchique, c'est-à-dire que le premier appelle les fonctions du second. Les fichiers présentés dans cette section concernent l'implémentation logicielle d'un égaliseur LTE-LMS de 8 coefficients.

Fichier main.cc

```
#include "data_read.h"
#include "equalizer_LMS.h"
#include <stdio.h>
#include <stdlib.h>
```

```
// Mettre MY_DEBUG a 0 pour eliminer les printf
//
#define MY_DEBUG   0
#define LENGTH     50000
#define TRAINING   1000
#define DELT_TRAIN 10000

int main() {

  bool mode = false;

/*****************************************************************/
/*          Declaration of variables of equalizers               */
/*****************************************************************/

  complex_short input_LTE, train_LTE;
  complex_short error_LTE, out_LTE;
  short nb_error_LTE = 0;
  FILE *in_LTE;

  // Data for LTE-LMS
  complex_short *xin_LTE;
  complex_short *dtrain_LTE;
  complex_short *w_LTE;

/*****************************************************************/
/*                    Initialization of equalizers               */
/*****************************************************************/


  in_LTE = open_file("vect_test.txt");
  init_equalizer_LMS( &xin_LTE, &w_LTE, &dtrain_LTE );


   for (int i = 0; i < LENGTH; i++) {

          if( (i%DELT_TRAIN) < TRAINING )
            mode = false;
          else
            mode = true;

/*****************************************************************/
/*                    Main functions for equaliztion             */
/*****************************************************************/

          read_value(in_LTE, &input_LTE, &train_LTE);
          out_LTE = equalizer_LMS( xin_LTE, w_LTE, dtrain_LTE, input_LTE, train_LTE,
                                   &error_LTE, &nb_error_LTE, mode );


#if MY_DEBUG

/*****************************************************************/
/*          Display the data of the equalizer for DEBUG          */
/*****************************************************************/
```

```
    printf( "------------ EQUALIZER LMS ------------------- \n\n");
    printf( "iteration : %d\n", i);
    for (short j = 0; j < N_TAPS ; j++)
            printf( "w_LTE[%d] = %d + %di \n", j, w_LTE[j].re, w_LTE[j].im);

    for (short j = 0; j < N_TAPS ; j++)
            printf( "xin_LTE[%d] = %d + %di \n", j, xin_LTE[j].re, xin_LTE[j].im);

    for (short j = 0; j < N_TAPS/2 ; j++)
            printf( "dtrain_LTE[%d] = %d + %di \n", j, dtrain_LTE[j].re, dtrain_LTE[j].im);

    printf( "input_LTE  = %d + %di \n", input_LTE.re, input_LTE.im);
    printf( "out_LTE = %d + %di \n", out_LTE.re, out_LTE.im);
    printf( "-----------------------------------------------\n\n");


#endif

    }       // end of main for

/*****************************************************************/
/* Display the weights of the equalizer and the number of error */
/*****************************************************************/

    printf( "------------ EQUALIZER LMS ------------------- \n\n");
    printf("Nombre d'erreurs total LTE : %d \n\n", nb_error_LTE );
    for (short j = 0; j < N_TAPS ; j++)
            printf( "w_LTE[%d] = %d + %di \n", j, w_LTE[j].re, w_LTE[j].im);
    printf( "-----------------------------------------------\n\n");

    free(xin_LTE);              // free the memory
    free(dtrain_LTE);
    free(w_LTE);
    fclose(in_LTE);             // close the file


    return 0;
}
```

Fichier equalizer_LMS.h

```
/*****************************************************************************
/
/       Fichier : equalizer_LMS.h
/
/       Description :   Fichier contenant les prototypes de fonction servant
/                       à l'implémentation d'un égaliseur LMS
/
/       Auteur  : Bruno Tanguay
/       Date    : 19 février 2004
/
/*****************************************************************************/
```

```
#ifndef _EQUALIZER_LMS_H_
#define _EQUALIZER_LMS_H_

#include "equalizer.h"


#define N_TAPS    8     // Nombre de coefficients du filtre
#define MU_LTE    2262  // Pas d'adaptation de l'égaliseur ( x 2^12) ( s_uuu.ffff_ffff_ffff)


/*****************************************************************************
/          Fonction : equalizer_LMS
/*****************************************************************************/

complex_short equalizer_LMS( complex_short* xin, complex_short* weigth,
          complex_short* data_train, complex_short input, complex_short train,
          complex_short* error, short* nb_error, bool mode );

void init_equalizer_LMS( complex_short** in, complex_short** weigth, complex_short** data_train );

#endif
```

Fichier equalizer_LMS.c

```
/*****************************************************************************
/
/          Fichier : equalizer_LMS.c
/
/          Description :     Fichier contenant les définitions de fonction servant
/                            à l'implémentation d'un égaliseur LMS
/
/
/          Auteur    : Bruno Tanguay
/          Date      : 27 janvier 2004
/
/*****************************************************************************/


#include "equalizer_LMS.h"
// Pour la fonction calloc (allocation dynamique de la mémoire)
#include <stdlib.h>
#include <stdio.h>
#define MY_DEBUG 0

/*****************************************************************************
/
/          Fonction : equalizer_LMS
/
/          Description :     Fonction principale servant à l'égalisation selon un
/                            algorithme LMS
/
/          Entrée    :
/          in             --> Pointeur à un ensemble de nombres complexes pour les données
/                              d'entrée du FIR
```

```
/       weigth      --> Pointeur à un ensemble de nombres complexes pour les
/                       coefficients
/       mu          --> Pas d'adaption pour la mise-à-jour des coefficients
/       data_train --> Donnée complexe servant pour l'entraînement
/       error       --> Erreur complexe résultant entre l'estimation et la décision
/       N_taps      --> Nombre entier représentant le nombre de taps du FIR complexe
/       nb_error   --> Nombre d'erreur effectuée jusqu'à maintenant
/       mode        --> Mode de poursuite ( false = entraînement, true = poursuite )
/       hold        --> Valeur booléene pour l'adaptation des coefficients
/                       (false = mise-à-jour des coeff. , true = maintient des coeff.)
/
/       Sortie :
/       y           --> Estimée de la donnée
/
/       Auteur    : Bruno Tanguay
/       Date      : 27 janvier 2004
/
/*****************************************************************************/


complex_short equalizer_LMS( complex_short* xin, complex_short* weigth,
        complex_short* data_train, complex_short input, complex_short train ,
        complex_short* error, short* nb_error, bool mode ){

        complex_short y;
        complex_short out;
        complex_short dec;
        complex_short err;

        for (short j = N_TAPS-1; j >0 ; j--) {
                xin[j] = xin[j-1];
        }

        for (short j = N_TAPS/2-1; j >0 ; j--) {
                data_train[j] = data_train[j-1];
        }

        xin[0]      = input;
        data_train[0] = train;

        // filtrage
        y = filter(xin, weigth, N_TAPS);

        //decision
#if BPSK
        dec = slicer_BPSK(y);
#endif

#if QPSK
        dec = slicer_QPSK(y);
#endif

        //compte les erreurs
        if( (dec.re != data_train[N_TAPS/2-1].re ) ){
                (*nb_error)++;
        }
```

```
        //mode d'entraînement
        if( mode == false){
          dec.re = data_train[N_TAPS/2-1].re;
          dec.im = data_train[N_TAPS/2-1].im;
        }

        //calcule l'erreur
        (*error).re  =  dec.re - y.re;
        (*error).im  =  dec.im - y.im;

#if MY_DEBUG
        printf("Error : %d + %di\n", (*error).re, (*error).im);
#endif

        //mise-à-jour des coefficients
        update_weigth(xin, weigth, MU_LTE, *error, N_TAPS);


        return y;
}


void init_equalizer_LMS( complex_short** in, complex_short** weigth, complex_short** data_train ){

  (*in)     = (complex_short*)calloc(N_TAPS, sizeof(complex_short));
  (*data_train) = (complex_short*)calloc(N_TAPS/2, sizeof(complex_short));
  (*weigth)    = (complex_short*)calloc(N_TAPS, sizeof(complex_short));

}
```

Fichier equalizer.h

```
#ifndef _EQUALIZER_H_
#define _EQUALIZER_H_

#include <stdlib.h>

#define BPSK     1
#define QPSK     0
#define QAM_16   0
#define QAM_64   0

#define BPSK_P   4096
#define BPSK_N   -4096

#define QPSK_P   4096
#define QPSK_N   -4096

#define QAM_P1   4096
#define QAM_P2   12288
#define QAM_P3   20480
#define QAM_N1   -4096
#define QAM_N2   -12288
#define QAM_N3   -20480
```

```
struct complex_int{
        int re;
        int im;
};

struct complex_short{
        short re;
        short im;
};

/*****************************************************************************
/       Fonction : filter
*****************************************************************************/

complex_short filter(complex_short* in, complex_short* weigth, const short nb_taps);

/*****************************************************************************
/       Fonction : slicer_BPSK
*****************************************************************************/

#if BPSK
complex_short slicer_BPSK(complex_short estimate);
#endif

/*****************************************************************************
/       Fonction : slicer_QPSK
*****************************************************************************/

#if QPSK
complex_short slicer_QPSK(complex_short estimate);
#endif

/*****************************************************************************
/       Fonction : slicer_QAM16
*****************************************************************************/

#if QAM_16
complex_short slicer_QAM16(complex_short estimate);
#endif

/*****************************************************************************
/       Fonction : slicer_QPSK
*****************************************************************************/

#if QAM_64
complex_short slicer_QAM64(complex_short estimate);
#endif


/*****************************************************************************
/       Fonction : update_weigth
*****************************************************************************/

void update_weigth(complex_short* in, complex_short* weigth, short mu,
                        complex_short err, const short nb_taps);
```

```
#endif
```

Fichier equalizer.c

```
#include "equalizer.h"

/*****************************************************************************
/
/       Fonction :  filter
/
/       Description : Fonction qui sert à calculer la sortie d'un FIR complexe
/
/       Entrée    :
/       in        --> Pointeur à un ensemble de nombres complexes pour les données
/                     d'entrée du FIR
/       weigth    --> Pointeur à un ensemble de nombres complexes pour les
/                     coefficients
/       N_taps    --> Nombre entier représentant le nombre de taps du FIR complexe
/
/       Sortie    :
/       out       --> Nombre complexe qui représente la sortie du FIR complexe
/
/       Auteur    :  Bruno Tanguay
/       Date      :  27 janvier 2004
/
*****************************************************************************/

complex_short filter(complex_short* in, complex_short* weigth, const short nb_taps)
{

        complex_int out;
        complex_short tr_out;
        out.re = 0;
        out.im = 0;

        for(short i = 0; i< nb_taps ; i++){
                out.re = out.re + (in[i].re*weigth[i].re) + (in[i].im*weigth[i].im);
                out.im = out.im + (in[i].im*weigth[i].re) - (in[i].re*weigth[i].im);
        }

        tr_out.re = (short)(out.re >> 16);
        tr_out.im = (short)(out.im >> 16);


#if MY_DEBUG
        printf("out : %d + %di\n", out.re, out.im);
        printf("tr_out : %d + %di\n", tr_out.re, tr_out.im);
#endif


        return tr_out;
}
```

```
/*****************************************************************************
/
/       Fonction : slicer_BPSK
/
/       Description : Fonction qui sert à effectuer une décision pour une modulation
/                     de type BPSK
/
/       Entrée    :
/       estimate -->      Nombre complexe qui représente un estimé de la donnée reçue
/                         (sortie du FIR complexe)
/
/       Sortie    :
/       dec       -->     Nombre complexe qui représente la décision
/
/       Auteur    : Bruno Tanguay
/       Date      : 27 janvier 2004
/
*****************************************************************************/

#if BPSK

complex_short slicer_BPSK(complex_short estimate){

        complex_short dec;

    if( estimate.re >= 0 ){
        dec.re = BPSK_P;
        dec.im = 0;
    }
    else{
        dec.re = BPSK_N;
        dec.im = 0;
        }

        return dec;
}

#endif


/*****************************************************************************
/
/       Fonction : slicer_QPSK
/
/       Description : Fonction qui sert à effectuer une décision pour une modulation
/                     de type QPSK
/
/       Entrée    :
/       estimate -->      Nombre complexe qui représente un estimé de la donnée reçue
/                         (sortie du FIR complexe)
/
/       Sortie    :
/       dec       -->     Nombre complexe qui représente la décision
/
/       Auteur    : Bruno Tanguay
/       Date      : 27 janvier 2003
/
```

```
/*****************************************************************************/

#if QPSK

complex_short slicer_QPSK(complex_short estimate){

        complex_short dec;

        if( estimate.re >= 0 ){
           dec.re = QPSK_P;
    }
    else{
           dec.re = QPSK_N;
        }

        if( estimate.im >= 0){
           dec.im = QPSK_P;
        }
        else{
           dec.re = QPSK_N;
        }


        return dec;
}

#endif


/*****************************************************************************
/
/       Fonction : slicer_QPSK
/
/       Description : Fonction qui sert à effectuer une décision pour une modulation
/                     de type QPSK
/
/       Entrée    :
/       estimate -->      Nombre complexe qui représente un estimé de la donnée reçue
/                         (sortie du FIR complexe)
/
/       Sortie    :
/       dec       -->     Nombre complexe qui représente la décision
/
/       Auteur    : Bruno Tanguay
/       Date      : 27 janvier 2004
/
/*****************************************************************************/

#if QAM_16

complex_short slicer_QPSK(complex_short estimate){

        complex_short dec;

        if( estimate.re >= 0 ){

           if( estimate.re > (QAM16_P1+QAM16_P2)/2 )
```

```
            dec.re = QAM16_P2;
          else
            dec.re = QAM16_P1;

    }
    else{

          if( estimate.re < (QAM16_N1+QAM16_N2)/2 )
            dec.re = QAM16_N2;
          else
            dec.re = QAM16_N1;
        }

        if( estimate.im >= 0 ){

          if( estimate.im > (QAM16_P1+QAM16_P2)/2 )
            dec.im = QAM16_P2;
          else
            dec.im = QAM16_P1;

    }else{

          if( estimate.im < (QAM16_N1+QAM16_N2)/2 )
            dec.im = QAM16_N2;
          else
            dec.im = QAM16_N1;
        }


        return dec;
}

#endif


/*****************************************************************************
/
/       Fonction : update_weigth
/
/       Description : Fonction qui sert à la mise-à-jour des coefficients de
/                     l'égalisateur selon l'algorithme LMS
/
/       Entrée    :
/       in        --> Pointeur à un ensemble de nombres complexes pour les données
/                     d'entrée du FIR
/       weigth    --> Pointeur à un ensemble de nombres complexes pour les
/                     coefficients
/       mu        --> Pas d'adaption pour la mise-à-jour des coefficients
/       err       --> Erreur complex_shorte calculée
/       N_taps    --> Nombre entier représentant le nombre de taps du FIR complex_shorte
/
/       Auteur    : Bruno Tanguay
/       Date      : 27 janvier 2004
/
*****************************************************************************/
void update_weigth(complex_short* in, complex_short* weigth, short mu,
                   complex_short err, const short nb_taps)
```

```
{
        complex_short fact_err;
        fact_err.re = (mu*err.re) >> 16;
        fact_err.im = (mu*err.im) >> 16;

#if MY_DEBUG
        printf("fact_err : %d + %di\n", fact_err.re, fact_err.im);
#endif

        for(short i = 0; i< nb_taps; i++){
                weigth[i].re = weigth[i].re + ((in[i].re*fact_err.re + in[i].im*fact_err.im) >> 16);
                weigth[i].im = weigth[i].im + ((in[i].im*fact_err.re - in[i].re*fact_err.im) >> 16);
        }

}
```

C.2 Programme pour le processeur Xtensa avec une unité spécialisée

Cette section comprend les fichiers suivants : main.cc, equalizer_LMS_tie_1mc.h et equalizer_LMS_tie_1mc.c. Les fichiers sont présentés dans un ordre hiérarchique, c'est-à-dire que le premier appelle les fonctions du second. Les fichiers présentés dans cette section concernent l'implémentation logicielle d'un égaliseur LTE-LMS 8 coefficients pour le processeur Xtensa avec 1 MC.

Fichier main.cc

```
#include "data_read.h"
#include "equalizer_LMS_tie_1mc.h"
#include "./tdk/include/xtensa/tie/eq_LMS_1mc_v4.h"
#include <stdio.h>
#include <stdlib.h>


// Mettre MY_DEBUG a 0 pour eliminer les printf
//
#define MY_DEBUG  0
#define LENGTH    50000
#define TRAINING  1000
#define DELT_TRAIN 10000

struct complex_int{
        int re;
        int im;
};
```

```
struct complex_short{
        short re;
        short im;
};


int main() {

  bool mode = false;

/*****************************************************************/
/*      Declaration of variables of equalizers                   */
/*****************************************************************/

  complex_short out_LTE_TIE;
  short nb_error_LTE_TIE = 0;
  FILE *in_LTE_TIE;

  // Data for LTE-LMS TIE
  int     xin_LTE_TIE = 0;
  short train_LTE_TIE = 0;


/*****************************************************************/
/*                Initialization of equalizers                   */
/*****************************************************************/

  in_LTE_TIE = open_file("vect_test.txt");
  initialize_eq_TIE();


   for (int i = 0; i < LENGTH; i++) {

        if( (i%DELT_TRAIN) < TRAINING )
          mode = false;
        else
          mode = true;

/*****************************************************************/
/*                Main functions for equaliztion                 */
/*****************************************************************/
        read_value_tie(in_LTE_TIE, &xin_LTE_TIE, &train_LTE_TIE);
        out_LTE_TIE = equalizer_LMS_TIE( xin_LTE_TIE, train_LTE_TIE, &nb_error_LTE_TIE, mode );


#if MY_DEBUG

/*****************************************************************/
/*      Display the data of the equalizer for DEBUG              */
/*****************************************************************/
   printf( "-------------  EQUALIZER LMS TIE ---------------\n\n");
   printf( "iteration : %d\n", i);
   printf( "w_LTE_TIE[0] = %d + %di \n", (short)(RFIRC0() >> 16), (short)(RFIRC0() & 0xFFFF));
   printf( "w_LTE_TIE[1] = %d + %di \n", (short)(RFIRC1() >> 16), (short)(RFIRC1() & 0xFFFF));
   printf( "w_LTE_TIE[2] = %d + %di \n", (short)(RFIRC2() >> 16), (short)(RFIRC2() & 0xFFFF));
   printf( "w_LTE_TIE[3] = %d + %di \n", (short)(RFIRC3() >> 16), (short)(RFIRC3() & 0xFFFF));
```

```
   printf( "w_LTE_TIE[4] = %d + %di \n", (short)(RFIRC4() >> 16), (short)(RFIRC4() & 0xFFFF));
   printf( "w_LTE_TIE[5] = %d + %di \n", (short)(RFIRC5() >> 16), (short)(RFIRC5() & 0xFFFF));
   printf( "w_LTE_TIE[6] = %d + %di \n", (short)(RFIRC6() >> 16), (short)(RFIRC6() & 0xFFFF));
   printf( "w_LTE_TIE[7] = %d + %di \n\n", (short)(RFIRC7() >> 16), (short)(RFIRC7() & 0xFFFF));
   printf( "FIR_D1 = %d + %di \n", (short)(RFIRD1() >> 16), (short)(RFIRD1() & 0xFFFF));
   printf( "FIR_D2 = %d + %di \n", (short)(RFIRD2() >> 16), (short)(RFIRD2() & 0xFFFF));
   printf( "FIR_D3 = %d + %di \n", (short)(RFIRD3() >> 16), (short)(RFIRD3() & 0xFFFF));
   printf( "FIR_D4 = %d + %di \n", (short)(RFIRD4() >> 16), (short)(RFIRD4() & 0xFFFF));
   printf( "FIR_D5 = %d + %di \n", (short)(RFIRD5() >> 16), (short)(RFIRD5() & 0xFFFF));
   printf( "FIR_D6 = %d + %di \n", (short)(RFIRD6() >> 16), (short)(RFIRD6() & 0xFFFF));
   printf( "FIR_D7 = %d + %di \n", (short)(RFIRD7() >> 16), (short)(RFIRD7() & 0xFFFF));
   printf( "FIR_D8 = %d + %di \n", (short)(RFIRD8() >> 16), (short)(RFIRD8() & 0xFFFF));
   printf( "DTRAIN3 = %d + %di \n", (short)RDTRAIN3() , 0);
   printf( "in_LTE_TIE  = %d + %di \n", (short)(xin_LTE_TIE >> 16), (short)(xin_LTE_TIE & 0xFFFF));
   printf( "out_LTE_TIE = %d + %di \n", out_LTE_TIE.re, out_LTE_TIE.im);
   printf( "-------------------------------------------------\n\n");
#endif

   }       // end of main for

/*********************************************************************/
/* Display the weights of the equalizer and the number of error */
/*********************************************************************/
   printf( "-------------  EQUALIZER LMS TIE ---------------\n\n");
   printf("Nombre d'erreurs total LTE tie: %d \n\n", nb_error_LTE_TIE );
   printf( "w_LTE_TIE[0] = %d + %di \n", (short)(RFIRC0() >> 16), (short)(RFIRC0() & 0xFFFF));
   printf( "w_LTE_TIE[1] = %d + %di \n", (short)(RFIRC1() >> 16), (short)(RFIRC1() & 0xFFFF));
   printf( "w_LTE_TIE[2] = %d + %di \n", (short)(RFIRC2() >> 16), (short)(RFIRC2() & 0xFFFF));
   printf( "w_LTE_TIE[3] = %d + %di \n", (short)(RFIRC3() >> 16), (short)(RFIRC3() & 0xFFFF));
   printf( "w_LTE_TIE[4] = %d + %di \n", (short)(RFIRC4() >> 16), (short)(RFIRC4() & 0xFFFF));
   printf( "w_LTE_TIE[5] = %d + %di \n", (short)(RFIRC5() >> 16), (short)(RFIRC5() & 0xFFFF));
   printf( "w_LTE_TIE[6] = %d + %di \n", (short)(RFIRC6() >> 16), (short)(RFIRC6() & 0xFFFF));
   printf( "w_LTE_TIE[7] = %d + %di \n", (short)(RFIRC7() >> 16), (short)(RFIRC7() & 0xFFFF));
   printf( "-------------------------------------------------\n\n");

   fclose(in_LTE_TIE);                  // close the file

   return 0;
}
```

Fichier equalizer_LMS_tie_1mc.h

```
/*****************************************************************************
/
/       Fichier : equalizer_LMS_tie_1mc.h
/
/       Description :   Fichier contenant les prototypes de fonction servant
/                       à l'implémentation d'un égaliseur LMS de 8 taps avec
/                       des instructions spécialisées TIE
/
/       Auteur  : Bruno Tanguay
/       Date    : 10 mars 2004
/
*****************************************************************************/
```

```
#ifndef _EQUALIZER_LMS_TIE_H_
#define _EQUALIZER_LMS_TIE_H_

#include "./tdk/include/xtensa/tie/eq_LMS_1mc_v4.h"

#define MU_LTE_TIE 2262   // Pas d'adaptation de l'égaliseur ( x 2^12) ( s_uuu.ffff_ffff_ffff)

/*****************************************************************************
/          Fonction : equalizer_LMS_TIE
*****************************************************************************/

complex_short equalizer_LMS_TIE( int in_tie, short data_train, short* nb_error, bool mode );

/*****************************************************************************
/          Fonction : initialize_eq_TIE
*****************************************************************************/

void initialize_eq_TIE();

#endif
```

Fichier equalizer_LMS_tie_1mc.c

```
/*****************************************************************************
/
/          Fichier : equalizer_LMS_tie_1mc.c
/
/          Description :      Fichier contenant les définitions de fonction servant
/                             à l'implémentation d'un égaliseur LMS de 8 taps avec
/                             des instructions spécialisées TIE
/
/          Auteur    : Bruno Tanguay
/          Date      : 10 mars 2004
/
*****************************************************************************/


#include "equalizer_LMS_tie_1mc.h"
// Pour la fonction calloc (allocation dynamique de la mémoire)
#include <stdlib.h>
#include <stdio.h>
#define MY_DEBUG 0


/*****************************************************************************
/
/          Fonction : equalizer_LMS_TIE
/
/          Description :      Fonction principale servant à l'égalisation selon un
/                             algorithme LMS avec des instructions spécialisées TIE
/
/          Entrée    :
/          in_tie       --> Donnée complexe pour le FIR (16 MSB --> partie réelle,
```

```
/                       16 LSB --> partie imaginaire)
/          data_train --> Donnée complexe servant pour l'entraînement
/          nb_error   --> Nombre d'erreur effectuée jusqu'à maintenant
/          mode       --> Mode de poursuite ( false = entraînement, true = poursuite )
/
/          Sortie :
/          y          --> Estimée de la donnée (complex_short)
/
/          Auteur   : Bruno Tanguay
/          Date     : 10 mars 2004
/
/*****************************************************************************/


complex_short equalizer_LMS_TIE( int in_tie, short data_train,
                                 short* nb_error, bool mode ){

          int out_tie;                  // estimé de la donnée en donnée TIE
          int dec_tie;                  // décision sur le symbole
          complex_short out;// estimé de la donnée en structure complexe

#if MY_DEBUG
          printf( "---------- EQUALIZER LMS TIE IN FUNCTION ---------\n\n");
          printf( "in_tie = %d + %di \n", (short)(in_tie >> 16), (short)(in_tie & 0xFFFF));
          printf( "data_train = %d + %di \n", data_train, 0);
          printf( "------------- BEFORE UPDATING WEIGTH -------------\n\n");
          printf( "w_LTE_TIE[0] = %d + %di \n", (short)(RFIRC0() >> 16), (short)(RFIRC0() & 0xFFFF));
          printf( "w_LTE_TIE[1] = %d + %di \n", (short)(RFIRC1() >> 16), (short)(RFIRC1() & 0xFFFF));
          printf( "w_LTE_TIE[2] = %d + %di \n", (short)(RFIRC2() >> 16), (short)(RFIRC2() & 0xFFFF));
          printf( "w_LTE_TIE[3] = %d + %di \n", (short)(RFIRC3() >> 16), (short)(RFIRC3() & 0xFFFF));
          printf( "w_LTE_TIE[4] = %d + %di \n", (short)(RFIRC4() >> 16), (short)(RFIRC4() & 0xFFFF));
          printf( "w_LTE_TIE[5] = %d + %di \n", (short)(RFIRC5() >> 16), (short)(RFIRC5() & 0xFFFF));
          printf( "w_LTE_TIE[6] = %d + %di \n", (short)(RFIRC6() >> 16), (short)(RFIRC6() & 0xFFFF));
          printf( "w_LTE_TIE[7] = %d + %di \n", (short)(RFIRC6() >> 16), (short)(RFIRC7() & 0xFFFF));
          printf( "FIR_D1 = %d + %di \n", (short)(RFIRD1() >> 16), (short)(RFIRD1() & 0xFFFF));
          printf( "FIR_D2 = %d + %di \n", (short)(RFIRD2() >> 16), (short)(RFIRD2() & 0xFFFF));
          printf( "FIR_D3 = %d + %di \n", (short)(RFIRD3() >> 16), (short)(RFIRD3() & 0xFFFF));
          printf( "FIR_D4 = %d + %di \n", (short)(RFIRD4() >> 16), (short)(RFIRD4() & 0xFFFF));
          printf( "FIR_D5 = %d + %di \n", (short)(RFIRD5() >> 16), (short)(RFIRD5() & 0xFFFF));
          printf( "FIR_D6 = %d + %di \n", (short)(RFIRD6() >> 16), (short)(RFIRD6() & 0xFFFF));
          printf( "FIR_D7 = %d + %di \n", (short)(RFIRD7() >> 16), (short)(RFIRD7() & 0xFFFF));
          printf( "FIR_D8 = %d + %di \n", (short)(RFIRD8() >> 16), (short)(RFIRD8() & 0xFFFF));
          printf( "DTRAIN3 = %d + %di \n", (short)RDTRAIN3() , 0);

#endif

          // Filtrage complexe
          out_tie = FIR(in_tie,0);
          out_tie = FIR(in_tie,1);
          out_tie = FIR(in_tie,2);
          out_tie = FIR(in_tie,3);
          out_tie = FIR(in_tie,4);
          out_tie = FIR(in_tie,5);
          out_tie = FIR(in_tie,6);
          out_tie = FIR(in_tie,7);
          out_tie = FIR(in_tie,8);
```

```
        // Décalage pour des données d'entraînement
        DTRAIN(data_train);

        //compte les erreurs
        if( (SBPSK(out_tie) >> 16) != RDTRAIN3()  )
                (*nb_error)++;

        // mode d'entraînement
        if( mode == false)
                dec_tie = TRAIN( out_tie );
        else
                dec_tie = SBPSK( out_tie );

        // Mise à jour des coefficients
        in_tie = ADAPT();
        UPDW(in_tie,0);
        UPDW(in_tie,1);
        UPDW(in_tie,2);
        UPDW(in_tie,3);
        UPDW(in_tie,4);
        UPDW(in_tie,5);
        UPDW(in_tie,6);
        UPDW(in_tie,7);
        UPDW(in_tie,8);

#if MY_DEBUG

        printf( "------------  AFTER UPDATING WEIGTH ----------\n\n");
        printf( "out_tie = %d + %di \n", (short)(out_tie >> 16), (short)(out_tie & 0xFFFF));
        printf( "mu_error = %d + %di \n", (short)(in_tie >> 16), (short)(in_tie & 0xFFFF));
        printf( "w_LTE_TIE[0] = %d + %di \n", (short)(RFIRC0() >> 16), (short)(RFIRC0() & 0xFFFF));
        printf( "w_LTE_TIE[1] = %d + %di \n", (short)(RFIRC1() >> 16), (short)(RFIRC1() & 0xFFFF));
        printf( "w_LTE_TIE[2] = %d + %di \n", (short)(RFIRC2() >> 16), (short)(RFIRC2() & 0xFFFF));
        printf( "w_LTE_TIE[3] = %d + %di \n", (short)(RFIRC3() >> 16), (short)(RFIRC3() & 0xFFFF));
        printf( "w_LTE_TIE[4] = %d + %di \n", (short)(RFIRC4() >> 16), (short)(RFIRC4() & 0xFFFF));
        printf( "w_LTE_TIE[5] = %d + %di \n", (short)(RFIRC5() >> 16), (short)(RFIRC5() & 0xFFFF));
        printf( "w_LTE_TIE[6] = %d + %di \n", (short)(RFIRC6() >> 16), (short)(RFIRC6() & 0xFFFF));
        printf( "w_LTE_TIE[7] = %d + %di \n", (short)(RFIRC7() >> 16), (short)(RFIRC7() & 0xFFFF));
        printf( "FIR_D1 = %d + %di \n", (short)(RFIRD1() >> 16), (short)(RFIRD1() & 0xFFFF));
        printf( "FIR_D2 = %d + %di \n", (short)(RFIRD2() >> 16), (short)(RFIRD2() & 0xFFFF));
        printf( "FIR_D3 = %d + %di \n", (short)(RFIRD3() >> 16), (short)(RFIRD3() & 0xFFFF));
        printf( "FIR_D4 = %d + %di \n", (short)(RFIRD4() >> 16), (short)(RFIRD4() & 0xFFFF));
        printf( "FIR_D5 = %d + %di \n", (short)(RFIRD5() >> 16), (short)(RFIRD5() & 0xFFFF));
        printf( "FIR_D6 = %d + %di \n", (short)(RFIRD6() >> 16), (short)(RFIRD6() & 0xFFFF));
        printf( "FIR_D7 = %d + %di \n", (short)(RFIRD7() >> 16), (short)(RFIRD7() & 0xFFFF));
        printf( "FIR_D8 = %d + %di \n", (short)(RFIRD8() >> 16), (short)(RFIRD8() & 0xFFFF));
        printf( "DTRAIN3 = %d + %di \n", (short)RDTRAIN3() , 0);
        printf( "dec_tie = %d + %di \n", (short)(dec_tie >> 16), (short)(dec_tie & 0xFFFF));
        printf( "-----------------------------------------------\n\n");
#endif

        // Conversion de la donnée TIE en structure complexe
        out.re = (out_tie >> 16);
        out.im = (out_tie & 0xFFFF);

        return out;
}
```

```
/*****************************************************************************
/
/          Fonction : initialize_eq_TIE
/
/          Description :      Fonction servant à l'initialisation de l'égaliseur LMS,
/                             définition du pas d'adaptation
/
/          Auteur   : Bruno Tanguay
/          Date     : 10 mars 2004
/
*****************************************************************************/

void initialize_eq_TIE(){

          WEQMU(MU_LTE_TIE);
}
```

C.3 Programme pour le processeur NIOS avec une unité spécialisée

Cette section comprend les fichiers suivants : copro_main_LTE_1mc.c, copro_equalizer_LTE_1mc.h et copro_equalizer_LTE_1mc.c. Les fichiers sont présentés dans un ordre hiérarchique, c'est-à-dire que le premier appelle les fonctions du second. Les fichiers présentés dans cette section concernent l'implémentation logicielle d'un égaliseur LTE-LMS 8 coefficients pour le processeur Nios avec 1 MC.

Fichier copro_main_LTE_1mc.c

```
/*****************************************************************************
/
/          Fichier : copro_main_LTE_1mc.c
/
/          Description :      Fichier contenant les définitions de fonction servant
/                             à l'implémentation d'un égaliseur LMS avec le NIOS
/                             (Coprocesseur 1 multiplieur complexe)
/
/
/          Auteur   : Bruno Tanguay
/          Date     : 10 juin 2004
/
*****************************************************************************/

#include "excalibur.h"
```

```
// Registres du coprocesseur
#define EQ_MU           0
#define EQ_ERROR        1
#define LTE_C0          2
#define LTE_C1          3
#define LTE_C2          4
#define LTE_C3          5
#define LTE_C4          6
#define LTE_C5          7
#define LTE_C6          8
#define LTE_C7          9
#define LTE_D0          10
#define LTE_D1          11
#define LTE_D2          12
#define LTE_D3          13
#define LTE_D4          14
#define LTE_D5          15
#define LTE_D6          16
#define LTE_D7          17
#define DTRAIN0 18
#define DTRAIN1 19
#define DTRAIN2 20
#define DTRAIN3 21

// Fonctions qui sert à afficher l'état des registres
void read_LTE_C();
void read_LTE_D();
void read_dtrain();


#include "copro_equalizer_LTE_1mc.h"
#include "copro_equalizer_LTE_1mc.c"


// Mettre MY_DEBUG a 0 pour eliminer les printf
//
#define EQ_LTE    1
#define MY_DEBUG  0

#define LENGTH    50000
#define TRAINING  1000
#define DELT_TRAIN 10000


int main() {

  int i;
  short j;
  char init = 0;
  int cnt = 0;

  volatile unsigned short nb_error_LTE = 0;
  unsigned int out_LTE;

  np_pio *dtrain = na_train_pio;
  np_pio *xin = na_data_in_pio;
```

```
  np_pio *add = na_rd_address_pio;

  init_equalizer();


   while(cnt < LENGTH){

     for ( i = 0; i < TRAINING; i++) {

              add->np_piodata = i;

        out_LTE = equalizer_LMS_blind( xin->np_piodata, dtrain->np_piodata, &nb_error_LTE);


           } // end of train for

     for ( i = TRAINING; i < DELT_TRAIN; i++) {

              add->np_piodata = i;

        out_LTE = equalizer_LMS_train( xin->np_piodata, dtrain->np_piodata, &nb_error_LTE);

     } // end of train for


           cnt += DELT_TRAIN;

}          // end of while


/*****************************************************************/
/* Display the weights of the equalizer and the number of error */
/*****************************************************************/


   printf( "------------ EQUALIZER LMS COPRO ----------\n\n");
   printf("Nombre d'erreurs total LTE : %d \n\n", nb_error_LTE );
   read_LTE_C();
   printf( "-------------------------------------------\n\n");


   printf("Nombre de symboles : %d \n", cnt);

   return 0;

}

// Fonction pour afficher les coefficients
void read_LTE_C(){

   printf( "w_LTE[0] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_C0) >> 16), (short)(nm_lte__pfx(6, 0, LTE_C0)
& 0xFFFF));
   printf( "w_LTE[1] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_C1) >> 16), (short)(nm_lte__pfx(6, 0, LTE_C1)
& 0xFFFF));
   printf( "w_LTE[2] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_C2) >> 16), (short)(nm_lte__pfx(6, 0, LTE_C2)
& 0xFFFF));
```

```
  printf( "w_LTE[3] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_C3) >> 16), (short)(nm_lte__pfx(6, 0, LTE_C3)
& 0xFFFF));
  printf( "w_LTE[4] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_C4) >> 16), (short)(nm_lte__pfx(6, 0, LTE_C4)
& 0xFFFF));
  printf( "w_LTE[5] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_C5) >> 16), (short)(nm_lte__pfx(6, 0, LTE_C5)
& 0xFFFF));
  printf( "w_LTE[6] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_C6) >> 16), (short)(nm_lte__pfx(6, 0, LTE_C6)
& 0xFFFF));
  printf( "w_LTE[7] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_C7) >> 16), (short)(nm_lte__pfx(6, 0, LTE_C7)
& 0xFFFF));

}

// Fonction pour afficher les données du filtre
void read_LTE_D(){

  printf( "FIR_D[0] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_D0) >> 16), (short)(nm_lte__pfx(6, 0, LTE_D0)
& 0xFFFF));
  printf( "FIR_D[1] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_D1) >> 16), (short)(nm_lte__pfx(6, 0, LTE_D1)
& 0xFFFF));
  printf( "FIR_D[2] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_D2) >> 16), (short)(nm_lte__pfx(6, 0, LTE_D2)
& 0xFFFF));
  printf( "FIR_D[3] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_D3) >> 16), (short)(nm_lte__pfx(6, 0, LTE_D3)
& 0xFFFF));
  printf( "FIR_D[4] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_D4) >> 16), (short)(nm_lte__pfx(6, 0, LTE_D4)
& 0xFFFF));
  printf( "FIR_D[5] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_D5) >> 16), (short)(nm_lte__pfx(6, 0, LTE_D5)
& 0xFFFF));
  printf( "FIR_D[6] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_D6) >> 16), (short)(nm_lte__pfx(6, 0, LTE_D6)
& 0xFFFF));
  printf( "FIR_D[7] = %d + %di \n", (short)(nm_lte__pfx(6, 0, LTE_D7) >> 16), (short)(nm_lte__pfx(6, 0, LTE_D7)
& 0xFFFF));

}

// Fonction pour afficher les données d'entraînement
void read_dtrain(){

  printf( "DTRAIN[0] = %d + %di \n", (short)(nm_lte__pfx(6, 0, DTRAIN0) >> 16), (short)(nm_lte__pfx(6, 0,
DTRAIN0) & 0xFFFF));
  printf( "DTRAIN[1] = %d + %di \n", (short)(nm_lte__pfx(6, 0, DTRAIN1) >> 16), (short)(nm_lte__pfx(6, 0,
DTRAIN1) & 0xFFFF));
  printf( "DTRAIN[2] = %d + %di \n", (short)(nm_lte__pfx(6, 0, DTRAIN2) >> 16), (short)(nm_lte__pfx(6, 0,
DTRAIN2) & 0xFFFF));
  printf( "DTRAIN[3] = %d + %di \n", (short)(nm_lte__pfx(6, 0, DTRAIN3) >> 16), (short)(nm_lte__pfx(6, 0,
DTRAIN3) & 0xFFFF));

}
```

Fichier copro_ equalizer_LTE_1mc.h

```
/*****************************************************************************
/
/          Fichier : copro_equalizer_LTE_1mc.h
```

```
/
/          Description :      Fichier contenant les prototypes de fonction servant
/                             à l'implémentation d'un égaliseur LMS avec le NIOS
/                             (Coprocesseur 1 multiplieur complexe)
/
/          Auteur    : Bruno Tanguay
/          Date      : 10 juin 2004
/
/*****************************************************************************/


#ifndef _COPRO_EQUALIZER_LTE_1MC_H_
#define _COPRO_EQUALIZER_LTE_1MC_H_

#define MU_LTE   10480   // Pas d'adaptation de l'égaliseur ( x 2^20)


/*****************************************************************************
/          Fonction : equalizer_LMS
/*****************************************************************************/

unsigned int equalizer_LMS_train( unsigned int input, unsigned short train,
                                  unsigned short* nb_error);

unsigned int equalizer_LMS_blin( unsigned int input, unsigned short train,
                                  unsigned short* nb_error);


void init_equalizer();

#endif
```

Fichier copro_ equalizer_LTE_1mc.c

```
/*****************************************************************************
/
/          Fichier : copro_equalizer_LTE_1mc.c
/
/          Description :      Fichier contenant les définitions de fonction servant
/                             à l'implémentation d'un égaliseur LMS avec le NIOS
/                             (Coprocesseur 1 multiplieur complexe)
/
/
/          Auteur    : Bruno Tanguay
/          Date      : 10 juin 2004
/
/*****************************************************************************/


#include "copro_equalizer_LTE_1mc.h"
#define MY_DEBUG 0


/*****************************************************************************
```

```
/
/       Fonction : equalizer_LMS_train
/
/       Description :   Fonction principale servant à l'égalisation (LMS) durant
/                       la phase d'entraînement
/
/       Entrée    :
/       input       --> Donnée complexe du symbole à égaliser
/       data_train --> Donnée complexe servant pour l'entraînement
/       nb_error  --> Nombre d'erreur effectuée jusqu'à maintenant
/
/       Sortie  :
/       y           --> Estimée de la donnée
/
/       Auteur    : Bruno Tanguay
/       Date      : 1 juin 2004
/
/*****************************************************************************/

unsigned int equalizer_LMS_train( unsigned int input, unsigned short train,
                                  unsigned short* nb_error){


        unsigned int y;
        unsigned int dec;
        unsigned int mu_error;

        // Filtrage
        nm_lte__pfx(0, input, 0);               // FIR
        nm_lte__pfx(0, input, 1);
        nm_lte__pfx(0, input, 2);
        nm_lte__pfx(0, input, 3);
        nm_lte__pfx(0, input, 4);
        nm_lte__pfx(0, input, 5);
        nm_lte__pfx(0, input, 6);
        nm_lte__pfx(0, input, 7);
        nm_lte__pfx(0, input, 8);
        y = nm_lte__pfx(0, input, 9);

        // Décalage des données d'entraînement
        nm_lte__pfx(1, train, 0);               // DTRAIN

        //compte les erreurs
        if( nm_lte__pfx(2, y, 0)  != nm_lte__pfx(6, 0, DTRAIN3)  )
                (*nb_error)++;

        // mode d'entraînement
        dec = nm_lte__pfx(3, y, 0);             // TRAIN

        // Calcul du facteur de correction
        mu_error = nm_lte__pfx(4, 0, 0);        // ADAPT

    // Mise à jour des coefficients
        nm_lte__pfx(5, mu_error, 0);            // UPDW
        nm_lte__pfx(5, mu_error, 1);
        nm_lte__pfx(5, mu_error, 2);
        nm_lte__pfx(5, mu_error, 3);
```

```
        nm_lte__pfx(5, mu_error, 4);
        nm_lte__pfx(5, mu_error, 5);
        nm_lte__pfx(5, mu_error, 6);
        nm_lte__pfx(5, mu_error, 7);
        nm_lte__pfx(5, mu_error, 8);
        nm_lte__pfx(5, mu_error, 9);

        return y;
}

/*****************************************************************************
/
/       Fonction : equalizer_LMS_blind
/
/       Description :   Fonction principale servant à l'égalisation (LMS) durant
/                       la phase de poursuite
/
/       Entrée  :
/       input       --> Donnée complexe du symbole à égaliser
/       data_train --> Donnée complexe servant pour l'entraînement
/       nb_error  --> Nombre d'erreur effectuée jusqu'à maintenant
/
/       Sortie  :
/       y           --> Estimée de la donnée
/
/       Auteur  : Bruno Tanguay
/       Date    : 1 juin 2004
/
*****************************************************************************/

unsigned int equalizer_LMS_blind( unsigned int input, unsigned short train,
                                  unsigned short* nb_error){


        unsigned int y;
        unsigned int dec;
        unsigned int mu_error;

        // Filtrage
        nm_lte__pfx(0, input, 0);               // FIR
        nm_lte__pfx(0, input, 1);
        nm_lte__pfx(0, input, 2);
        nm_lte__pfx(0, input, 3);
        nm_lte__pfx(0, input, 4);
        nm_lte__pfx(0, input, 5);
        nm_lte__pfx(0, input, 6);
        nm_lte__pfx(0, input, 7);
        nm_lte__pfx(0, input, 8);
        y = nm_lte__pfx(0, input, 9);

        // Décalage des données d'entraînement
        nm_lte__pfx(1, train, 0);               // DTRAIN

        //compte les erreurs
        if( nm_lte__pfx(2, y, 0)  != nm_lte__pfx(6, 0, DTRAIN3)  )
                (*nb_error)++;
```

```
        // mode d'entraînement
        dec = nm_lte__pfx(2, y, 0);             // SBSPK

        // Calcul du facteur de correction
        mu_error = nm_lte__pfx(4, 0, 0);        // ADAPT

    // Mise à jour des coefficients
        nm_lte__pfx(5, mu_error, 0);            // UPDW
        nm_lte__pfx(5, mu_error, 1);
        nm_lte__pfx(5, mu_error, 2);
        nm_lte__pfx(5, mu_error, 3);
        nm_lte__pfx(5, mu_error, 4);
        nm_lte__pfx(5, mu_error, 5);
        nm_lte__pfx(5, mu_error, 6);
        nm_lte__pfx(5, mu_error, 7);
        nm_lte__pfx(5, mu_error, 8);
        nm_lte__pfx(5, mu_error, 9);

        return y;
}


/*****************************************************************************
/
/       Fonction : init_equalizer
/
/       Description :   Fonction principale servant à l'initialisation de
/                       l'égaliseur (LMS)
/
/
/       Auteur  : Bruno Tanguay
/       Date    : 1 juin 2004
/
*****************************************************************************/

void init_equalizer(){

        nm_lte__pfx(7, (MU_LTE << 16), EQ_MU);  // WRITE EQ_MU

        nm_lte__pfx(7, 0, LTE_C0);
        nm_lte__pfx(7, 0, LTE_C1);
        nm_lte__pfx(7, 0, LTE_C2);
        nm_lte__pfx(7, 8192, LTE_C3);
        nm_lte__pfx(7, 0, LTE_C4);
        nm_lte__pfx(7, 0, LTE_C5);
        nm_lte__pfx(7, 0, LTE_C6);
        nm_lte__pfx(7, 0, LTE_C7);

        nm_lte__pfx(7, 0, LTE_D0);
        nm_lte__pfx(7, 0, LTE_D1);
        nm_lte__pfx(7, 0, LTE_D2);
        nm_lte__pfx(7, 0, LTE_D3);
        nm_lte__pfx(7, 0, LTE_D4);
        nm_lte__pfx(7, 0, LTE_D5);
        nm_lte__pfx(7, 0, LTE_D6);
        nm_lte__pfx(7, 0, LTE_D7);
}
```

ANNEXE D OPTIMISATION LOGICIELLE

Cette annexe présente les principales techniques d'optimisation logicielles. Elle est constituée de 12 sections différentes, chacune des sections concernent une technique d'optimisation en particulier. Pour chaque technique d'optimisation, une brève description et un exemple sont fournis. Il est à noter que la plupart des optimisations sont réalisées par le compilateur, mais il est recommandé cependant que le programmeur utilise ces techniques de codage afin de s'assurer que le code compilé soit optimal (il ne faut pas supposer que le compilateur va effectuer toutes les optimisations). De plus, quelques trucs pour optimiser l'exécution d'un programme sont exhibés à la fin cette annexe.

D.1 Élimination de code inutile

(« Dead Code Elimination »)

Description : Cette technique consiste à éliminer les sections de code dont l'exécution n'effectue rien d'utile ou qui ne sera jamais exécuté. Elle permet de réduire la largeur du code.

Exemple :

Code Original	Code Transformé
x = 0;	x = 0;
y = 1;	y = 1;
if(y) x =0;	

D.2 Propagation de constante

(« Constant propagation »)

Description : Cette technique consiste à propager une constante à travers le code. Elle permet de réduire le temps de calcul et le nombre d'accès en mémoire. De plus, elle permet d'évaluer des expressions conditionnelles à la compilation.

Exemple :

Code Original	Code Transformé
int x, y, z;	int x, y, z;
x = 1;	y = 1;
z = x* y;	z = 1 * y;

D.3 Fusion de constante

(« Constant Folding »)

Description : Cette technique consiste à fusionner des constantes ensemble afin de créer une nouvelle constante.

Exemple :

C_in, D_in sont des constantes.

Code Original	Code Transformé	
int x, a, b;	int x, a, b;	
x = C_in*(a*b^2) ;	x = E_in*(a*b^2) ;	// E_in = C_in/D_in
x = x/D_in;		

D.4 Propagation de copie

(« Copy propagation »)

Description : Cette technique consiste à concentrer l'accès sur une variable, ce qui facilite au compilateur à déterminer quelle variable doit être emmagasinée en registre.

Exemple :

Code Original	Code Transformé
int x, y, z;	int x, y, z;
y = x;	y = x;
z = y;	z = x;

D.5 Élimination de sous expression commune

(« Common subexpression elimination »)

Description : Cette technique consiste à emmagasiner le résultat d'une sous expression et de la propager à travers les autres expressions. Elle permet d'éviter de recalculer une expression calculée antérieurement, ce qui réduit le temps d'exécution.

Exemple :

Code Original

```
int a,b,c,x,y,z;
x = a /b;
y = a/b + 3;
z = a/b*c;
```

Code Transformé

```
int a,b,c,x,y,z, tmp;
tmp = a /b;
x = tmp;
y = tmp + 3;
z = tmp*c;
```

D.6 Déroulement de boucle

(« Loop Unrolling »)

Description : Cette technique consiste à dérouler la boucle un certain nombre d'itération afin de réduire le coût pour les instructions de la boucle.

Exemple :

Code Original

```
for( i = 0; i < 100; i++){
        a[i] = b[i] + 5;
}
```

Code Transformé

```
for( i = 0; i < 100; i += 4){
        a[i] = b[i] + 5;
        a[i + 1] = b[i+1] + 5;
        a[i + 2] = b[i+2] + 5;
        a[i + 3] = b[i+3] + 5;
}
```

D.7 Déplacement de code invariant

(« Invariant Code Motion »)

Description : Cette technique consiste à déplacer du code à l'intérieur d'une boucle qui ne change pas d'une itération à l'autre (invariant) vers l'extérieur de la boucle.

Exemple :

Code Original	Code Transformé
int x, i, a;	int x, i, a;
for(i = 0; i < 100; i++){	if(x == 1) a = 5;
if(x == 1) a = 5;	for(i = 0; i < 100; i++){
do_something();	do_something();
}	}

D.8 Simplification mathématique et logique

Description : Cette technique consiste à remplacer des expressions complexes par des plus simples qui réalisent la même fonctionnalité.

Exemple :

Code Original	Code Transformé	
y = x/4;	y = x >> 2;	Division ou multiplication par 2^n
y = x %16;	y = x & 0x0F;	Modulo par une puissance de 2
y = x^3;	y = x*x*x;	Puissance
y = a*m + b*m;	y = (a+ b)*m	Factorisation
if((x & 2) || (x & 8)) {...}	if(x &10) {...}	
if(a==b && c==d && e==f) {...}	if(((a-b)|(c-d)|(e-f)) == 0) {...}	
if((x==1) || (x==2) || (x==4) || (x==8) || ...)	if(x&(x-1)==0 &&x!=0)	
	if(x&(x-1)==0 &&x!=0)	
if((x >= 0) && (x < 8) && (y >= 0) && (y < 8)) {...}	if(((unsigned) (x | y)) <8) {...}	
if((x==2) || (x==3) || (x==5) || (x==7) || (x==11) || (x==13) || (x==17) || (x==19)) {...}	if((1<<x) & ((1<<2)|(1<<3)|(1<<5)|(1<<7) |(1<<11)|(1<<13)|(1<<17)|(1<<19))) {...}	

D.9 Alignement de fonction

(« Inlining Functions »)

Description : Cette technique consiste à remplacer les appels de fonction par le corps de la fonction (Comme macro). Elle permet de réduire le temps d'exécution en évitant les appels de fonctions, mais elle augmente la largeur du code.

Exemple :

Code Original	Code Transformé

```
int comp(int a, int b){
        return (b / a) – (b*a);
}
```

```
inline int comp(int a, int b){
        return (b / a) – (b*a);
}
```

D.10 Interchangement de boucle

(« Loop Interchange »)

Description : Cette technique permet d'interchanger l'ordre d'imbrication de deux boucles. Celle-ci permet de réduire le nombre de « cache miss », donc le temps d'exécution.

Exemple :

Code Original

```
for( i = 0; i < 100; i++){
   for( j = 0; j < 15; j++){
        a[j][i] = b[j][i] + c[j][i];
   }
}
```

Code Transformé

```
for( j = 0; j < 15; j++){
   for( i = 0; i < 100; i++){
        a[j][i] = b[j][i] + c[j][i];
   }
}
```

D.11 Fusion de boucle

(« Loop fusion »)

Description : Cette technique peut être appliqué à deux boucles adjacentes dont la valeur du compteur initial et final est identique. Elle permet de réduire le temps d'exécution en éliminant les instructions de bouclage redondantes. De plus, cette optimisation réduit la largeur du code.

Exemple :

Code Original

```
for( i = 0; i < 100; i++){
        do_something1();
}
for( i = 0; i < 100; i++){
        do_something2();
}
```

Code Transformé

```
for( i = 0; i < 100; i++){
        do_something1();
        do_something2();
}
```

D.12 Accès par bloc

(« Blocking »)

Description : Cette technique permet de réduire le nombre de « data miss ». Cette technique doit être utilisée avec de larges tableaux multidimensionnels qui sont accédé à la fois par colonne et par rangée. Elle consiste à diviser les tableaux en sous matrice.

Exemple :

Code Original

```
for(i = 0; i <N; i++){
  for(j = 0; j <N; j++){
    r = 0;
    for(k = 0; k <N; k++)
      r = r + y[i][k]*z[k][j];
    x[i][j] = r;
  }
}
```

Code Transformé

```
for(jj = 0; jj <N; jj = jj +B)
  for(kk = 0; kk <N; kk = kk + B)
    for(i = 0; i <N; i++)
      for(j = jj; j < min(jj+B,N); j++){
        r = 0;
        for(k = kk; k < min(kk+B,N) ; k++)
          r = r + y[i][k]*z[k][j];
        x[i][j] = x[i][j] + r;
      }
```

Trucs :

- Ordre inverse, décrémenter le compteur vers 0, ça évite une comparaison
- Il est parfois avantageux d'utiliser des LUT plutôt que des gros switch ou de générer la valeur
- Éviter les variables globales, static et volatile
- Spécifier les données unsigned lorsque celles-ci ne sont jamais négatives
- Éviter l'allocation dynamique
- Éviter les divisions, modulos, racine carrée
- Taille des tableaux en puissance de 2 (+ facile à calculer l'adresse)
- Éviter les pointeurs autant que possible
- Briser les longues chaînes de switch ou if en plus petites imbriquées
- Les accès de tableau multidimensionnel doivent être accédé par les indices de droite en premier.
- Le passage des structures doit être effectué par un pointeur
- Activer les optimisations du compilateur
- Lorsque possible, effectuez un break dans la boucle plutôt de continuer les itérations suivantes.